Under consideration for publication in Formal Aspects of Computing

A Dynamic Logic for Deductive Verification of Multi-threaded Programs

Bernhard Beckert and Vladimir Klebanov

Institute for Theoretical Informatics, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
E-mail: {beckert,klebanov}@kit.edu

Abstract. We present MODL, a Dynamic Logic and a deductive verification calculus for a core Java-like language that includes multi-threading. The calculus is based on symbolic execution. Even though we currently do not handle non-atomic loops, employing the technique of symmetry reduction allows us to verify systems without limits on state space or thread number.

We have instantiated our logic for (restricted) multi-threaded Java programs and implemented the verification calculus within the KeY system. We demonstrate our approach by verifying a central method of the `StringBuffer` class from the Java standard library in the presence of unbounded concurrency.

Keywords: Multi-threading; Deductive verification; Symbolic execution; Symmetry reduction; Input/output reasoning

1. Introduction

1.1. Motivation and Goals

Verification of concurrent systems 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 [RDH03, HP00]. Nonetheless, for verification problems that are data-centric or that involve an unbounded number of threads, deductive verification offers advantages. The properties that we deal with can, in general, neither be expressed in temporal logic nor verified with a model checker.

We present a Dynamic Logic and a deductive verification calculus for a core Java-like language that includes multi-threading. In parallel to Object-oriented Dynamic Logic (ODL) [BP06], which captures the essence of object-orientation in a small language, we have called our logic MODL—Multi-threaded Object-oriented Dynamic Logic. In Section 6.2, we describe an (implemented) mapping from Java programs satisfying our requirements (cf. next section) to MODL. The mapping is such that the Java program and its counterpart in MODL perform the same state transition. If the MODL program can be verified, then the original Java program is correct as well.

Our aim has been to design a program logic that

- reflects the properties of thread-based concurrency in an intuitive manner
- has a sound and (relatively) complete calculus
- employs only sound and transparent for the user abstractions
- poses no bounds on the state space or thread number

Correspondence and offprint requests to: Vladimir Klebanov, e-mail: klebanov@kit.edu

- allows reasoning about complex properties of the scheduler, 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 do handle 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 the significance of these restrictions in the next section.

Our calculus has been implemented in the KeY system [BHS07], which has been successfully used for verification of non-concurrent Java programs. An application of our method to verify one of the most commonly used pieces of production Java code in the presence of unbounded concurrency is described towards the end.

1.2. Restrictions of MODL and Achieved Java Coverage

Not all Java programs can currently be modelled faithfully (and thus verified) with MODL. In the following, we describe the abilities and limitations of MODL from this point of view. The restrictions stem from the simplified model of the scheduler, and we believe that they can be overcome in the future by elaborating the model.

Sequential coverage The MODL programming language we define in the following is intentionally minimalistic. In the implementation, though, we benefit from the 100% Java Card coverage of the KeY calculus. This includes full support for dynamic object creation (with static initialisation), efficient aliasing treatment, Java-faithful arithmetic, etc. All of these features can be reduced to MODL, resp. used in verification of concurrent programs.

One area where there is currently a gap between the concurrent and the sequential calculus is exception handling. The concurrent proof system is sound but incomplete in this regard. Exceptions are always detected, but once thrown they cannot be caught. The calculus treats the whole program as non-terminating in this case.

No dynamic thread creation (but unbounded multi-threading) In Java, threads are created and (to some extent) controlled via instances of the `java.lang.Thread` class. The only thread creation mechanism MODL currently provides is a possibility 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").

This limitation 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.

No thread identities in programs MODL threads can communicate by exchanging data through the shared heap and by mutual exclusion primitives (synchronized methods and blocks in Java, lock acquisition and release operations in MODL). We have performed initial experiments with condition variables (`wait()`/`notify()` in Java), but the rules are not yet mature enough to be presented here.

All of these communication means have in common that they work without identifying threads via corresponding instances of `java.lang.Thread` (as mentioned above for thread creation). Indeed, we currently do not provide for a connection between such instances and MODL threads. A programmer, thus, cannot invoke thread-controlling methods of the `Thread` instance, the most important being `t.interrupt()` and `t.join()`. We believe that this limitation prevents us from verifying only a small fraction of interesting code.

No non-atomic loops Finally, we require all loops to be atomic. This is the biggest limitation currently. It means that the programmer has to ensure that no interference with other threads is possible while a loop runs. For Java programs, compliance with this restriction must be proven to maintain soundness. We can check an overapproximation of this property as shown later on (⇒ Sect. 8.3). Detailed technical reasons for this restriction are given in Note 3.2, Section 3.6 when we describe our model of the scheduler. In the cases when the non-atomic loop is on the top level of a program (e.g., in servers), it is possible to cut the loop and verify its body under execution by an unlimited number of threads.

Java Memory Model 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 guarantees sequential consistency only for

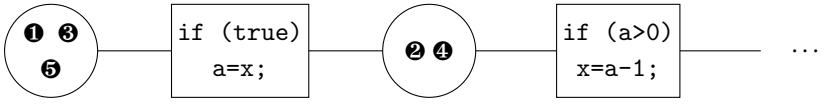


Fig. 1. Graph representation of a multi-threaded program {❶ ❸ ❺}`a=x; if (a>0)` {{❷ ❹}`x=a-1; ...`}

data-race-free programs. We have developed and implemented a calculus extension to check programs for absence of data races [Kle09].

1.3. Main Idea of the Proposed Logic and Proof System

The logic MODL Unsurprisingly, MODL is a close relative of Java Card DL, the sequential KeY logic [BHS07]. It has the DL-customary modal operators $\langle p\rangle\phi$ and $[p]\phi$ referring to the total and partial correctness of a program p w.r.t. the postcondition ϕ. The biggest difference lies in the programs: multi-threaded programs require a different representation than sequential ones. We use the CFG-style program model of Keller [Kel76], who has defined "parallel programs" as

a bipartite directed graph, the nodes of which are divided into

- place nodes: representing points at which an instruction pointer of a processor may dwell,
- transitions nodes: representing a class of transitions, each denoting an event which corresponds to the execution of a particular instruction.

In our case, the role of place nodes is played by set-valued control variables, which are part of the state and contain thread ids (collectively we also call them a thread configuration). The transition nodes are given by Java-like statements, which appear as "program text" inside the modal operator.

Execution of a program corresponds to the movement of thread id "tokens," while the program text remains unaltered. The movement is accompanied by a corresponding change in data state. It is clear that programs can behave differently depending on the thread scheduling. The natural question is how to model the scheduler?

With a purely nondeterministic scheduling, we have no choice but to perform (a prohibitively large number of) case distinctions in the calculus. Meta-level efforts would then be necessary to prune the proof search space and get a grip on the complexity (this is an approach taken in [BDRS02]). Instead, we opt for an underspecified deterministic scheduler, and express its decisions explicitly on the object level by means of a partially specified scheduling function.

Such a design gives our concurrent programs (surprisingly maybe) a deterministic semantics: a program started in a given state has at most one final state, just as is the case with sequential Java programs. The main advantage is the much stronger control over granularity of reasoning. By reifying the scheduler in the logic, we gain the power to express complex scheduling properties for demanding cases, but still can tackle simple problems with relatively little effort. Furthermore, we retain beneficial logical properties, like $\langle p\rangle\phi \rightarrow [p]\phi$.

A calculus for MODL To prove theorems of the above logic we have developed a sequent-style calculus. The calculus performs symbolic execution of programs—a method (going back to [Kin76]) that ensures good understandability of the process for the user. As far as we know, this work is the first application of symbolic execution to full functional verification of multi-threaded programs foregoing a bound on the number of threads or explored program states.

In principle, the calculus explores the behaviour of a concurrent program by building all possible thread interleavings. Done naively, such an approach is doomed to failure due to state explosion; it is also inapplicable to systems with an unbounded number of threads. Our calculus can effectively perform such exploration by employing symmetry reductions that merge a large number of structurally identical interleavings. This is efficiently possible for the considered language fragment and produces a feasible number of cases (even in the presence of unbounded concurrency). Further efficiency gains are possible from appropriate program and proof modularisation.

By means of symbolic execution, the calculus reduces assertions about programs to assertions about data types and permutations, which encapsulate the scheduler decisions and hide symmetric schedulings. In the desirable case that the program is scheduling-independent[1] the permutations can be removed from the correctness assertions by application of standard algebraic lemmas. When also the remaining assertions (now without permutations) can be discharged, then the program is fully correct w.r.t. its functional specification.

[1] Scheduling independence means here that the program's final state always satisfies the specification, in spite of possibly different intermediate states taken in different runs.

2. Related Work

Classical approaches to deductive verification of concurrent programs One of the first deductive verification methods for concurrent programs was the partial correctness proof method of Ashcroft [Ash75] and Keller [Kel76], incorporating a CFG-like program formalism and an induction principle. The principle is to show that every atomic statement preserves a global invariant. Of course, such global invariants can quickly become unwieldy without modularisation. Nonetheless, these early works contain many seminal insights into the inner workings of concurrent programs.

Another classical method is due to Owicki and Gries [OG76] and builds on Hoare Logic for sequential programs. The method combines a proof of local (i.e., sequential) correctness with a non-interference check. The latter establishes that assumptions used throughout the proof of local correctness are not destroyed if the scheduler chooses to interleave execution with other threads. This leads to proof size that is quadratic in the number of statements. The method is not compositional. We have implemented an Owicki-Gries-style proof system for a fragment of Java in KeY [Kle04]. Further modern adaptations of the method are described in the next section.

A revolutionary step towards compositional verification of concurrent programs was the rely-guarantee method of Jones [Jon81]. The method introduces for each thread two predicates: *rely* and *guarantee*. In contrast to assertions or postconditions, these predicates range not over states but over pairs of consecutive states. The proof method consists in showing that every step of a thread satisfies its guarantee obligation assuming that every step of the environment satisfies the rely assumption. The rely assumption in its turn is composed from the guarantee obligations of other threads. The method is compositional and the proof size is linear in the number of threads. The difficulty resides in summarising the behaviour of a thread in one transitive predicate.

Deductive verification of multi-threaded Java programs Several deductive calculi for (different fragments of) sequential Java exist [JP01, PHM99, vO01, ZKR08, MPMU04]. In contrast, the only implemented deductive verification system for multi-threaded Java existing to date is—to our knowledge—Verger [AdBdRS05]. The calculus is an adaptation of the Owicki-Gries method to Java, incorporating a proof method for CSP in order to reason about method calls as message passing. The system generates verification conditions from programs augmented with auxiliary variables and annotated with Hoare-style assertions. The verification conditions are subsequently discharged in PVS. The system has good concurrent language coverage.

A recent and more accessible formulation is [dB07], which replaces the CSP calculus with proof theory of recursive procedures.

Separation Logic is another extension of Hoare Logic with operators for reasoning about resource access, which allows for greater modularity of reasoning. The logic has also been extended to handle Java and concurrency, and the latest development is a "marriage" of rely-guarantee and Separation Logic [VP07]. The latter promises better modularity in dealing with rely and guarantee predicates.

Temporal logics 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 [MP91]). 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.

Concurrent Dynamic Logic The only Dynamic Logic for a programming language incorporating concurrency is—to our knowledge—the Concurrent Dynamic Logic (CDL) by Peleg [Pel87b]. He notes, however, that this particular logic "suffers from the absence of any communication mechanisms; processes of CDL are totally independent and mutually ignorant." Peleg gives two extensions of CDL with interprocess communication in [Pel87a]: 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). The issue of a calculus or program verification in general is not touched.

Model checkers Formal analysis of *concurrent* systems has traditionally been—with a few exceptions—the domain of model checking tools. This holds also for Java programs, where several very successful model checking frameworks have been established. Prominent model checkers for Java programs are Bogor [RDH03] and Java PathFinder [HP00].

Modern software model checkers can check not only temporal but also functional properties. They employ very clever optimisations and abstractions and can verify programs of substantial size. Many of the employed techniques—like symmetry reduction—are sound and do not come at the price of missed errors. Still, to guarantee termination of

the model checking process, a finite system model is required. For concurrent programs, finiteness must be achieved either by unsound abstraction (bounding the length of explored executions, number of threads, context switches, heap configurations, etc.) or through a loss of precision.

As such, model checking is very useful for detecting bugs but is not intended for full functional verification. An interesting extension of model checking and a close relative of our approach is [KPV03], which allows symbolic execution of multi-threaded Java programs in a model checker by means of program instrumentation. The instrumentation lifts the program data to symbolic values (delegating reasoning to an external decision procedure), but lets the model checker handle the concrete threads. The framework can "prove correctness for programs that have finite execution trees and have decidable data constraints."

A comprehensive control flow model of Java concurrency is given in [DRB02]. The authors use a variant of Petri nets to model the control flow of concurrent programs. The nets are specifically tailored to treat the "partially non-blocking rendez-vous" nature of Java's `wait()`/`notify()` mechanism. The authors do not perform functional verification but have built a model checker that can check safety properties expressed in terms of control flow. Their Petri net representation is conceptually close to ours, though we use full programs as transitions.

Yahav [Yah01] describes a system for verifying safety properties of multi-threaded Java-like programs. The system (implemented in the TVLA framework) is an instance of symbolic on-the-fly model checking, where first-order logical structures are used to represent states of the program. It can cope with an unbounded number of allocated objects by building conservative abstract descriptions of (multiple) program states in 3-valued logic. Also, in the above paper, symmetry reduction is mentioned and the author reports having obtained interesting results for an unbounded number of threads.

Static verifiers Another broad category of verification systems for concurrent programs are static verifiers. Static verifiers are tools that can automatically check program properties by sufficiently approximating the program semantics. Many static verifiers allow the users to improve the quality of the approximation by adding annotations to the code.

Per design, static verifiers are not geared towards input-output reasoning. They are—in most cases—also neither sound nor complete. Still, such tools are very useful for automated detection of concurrency-related problems in practice. There is also a big potential in combining static verification systems with systems for full-functional verification.

A prominent representative of this class of tools is ESC/Java [FLL$^+$02], an extended static checker for many types of properties. On the concurrency side this includes inter-thread escape analysis, race condition detection, deadlock detection, etc. There are also a number of dedicated static analysis tools for race condition detection. One of them is Houdini/rcc [AFF06], which is based on an elaborate type system.

Such tools are aimed to check that access to object fields is guarded by locks and that all threads adhere to a consistent locking policy. This check can be easy if the object fields are protected by the lock associated with the object itself or a dedicated object referenced by a final static field. More elaborated locking schemes might require user annotation or are beyond the scope of the tools.

A class of its own in this category is the SPEC# system, which (in its derivative SpecLeuven) incorporates a "static verifier" for a concurrent object-oriented language [JSPS06]. For one, verification with SPEC# guarantees the absence of data races and deadlocks. It also guarantees compliance of the program with programmer-provided method contracts and object invariants.

A very interesting body of research has been produced by Greenhouse and Scherlis [GS02]. The authors have developed an annotation language to specify many important aspects of multi-threaded programs together with a tool suite to statically check them. The annotations include:

- effects (an upper bound on state a method reads and writes)
- aliasing intent. Unaliased data can be reasoned about sequentially
- locking intent. Programmers can associate locks with regions of state; the tool verifies that state is accessed only when the appropriate lock is held. Programmers can also declare that a method requires that a particular lock be held by the caller
- concurrency policy. Programmers can specify methods that can be safely executed concurrently.

The authors also make it plausible that for lock-based programs, concurrency policy combined with models of locking intent can be a surrogate for representation invariants.

3. Syntax of MODL

We start with a very general formalism that is quite close to the "machine" semantics. The usefulness of this logic is not so much in its suitability for verification (this will be addressed in Section 5), but in formalising basic concepts of thread-based concurrency. We define the syntax and semantics of a multi-threaded Java-like programming language and a Dynamic Logic for reasoning about it. Along the way we introduce such concepts as thread configurations, shared and thread-local data, and a deterministic scheduler model.

3.1. Threads and Multi-threaded Programs

The concurrent programming language that we consider is very close to a fragment of multi-threaded Java. Its basic constructs are assignments, if-then-else statements, while-loops, Java-like concurrency primitives (lock acquisition and release), but also atomic blocks. Several threads can execute a program concurrently. Thus, in contrast to the sequential programs in KeY, a concurrent program is a passive template "without life," until a thread configuration is added. A thread configuration is a part of the program state describing which threads are executing the program. Threads are given a unique identifier, conventionally called *thread id* (tid), which is a term of type Thread; they are in fact identified with this identifier. In the following, we will denote the carrier set of Thread as $\mathscr{T}$.

In addition to concurrent programs, we also use sequential MODL programs. A sequential program is, roughly, a concurrent program executed by a single thread. The executing thread is explicitly identified in thread-local variables of the program. This explicit thread identifier is also the major difference between sequential MODL programs and sequential Java programs as formalised in KeY.

Modelling locking in our programming language A Java thread can enter a synchronized method or block only after successfully acquiring the lock of the object synchronized on. When a thread's control flow leaves the synchronization scope, the involved lock is automatically released. Locks are reentrant: if a thread already possesses a certain lock, a repeated acquisition of the same lock succeeds, increasing the locking depth.

To make lock acquisition and release explicit, we extend the `Object` class with two "ghost" methods: `public void <lock>()` and `public void <unlock>()`. Code marked as synchronized is automatically surrounded by invocations of these methods during the unfolding stage (⇒ Sect. 6.2). To keep track of locking state, we also declare two ghost fields per object: `<lockedby>` of type Thread (the identity of the thread holding the object's lock) and `int <lockcount>` (the locking depth).

3.2. 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; in contrast, the interpretation of non-rigid function symbols may differ from state to state. Moreover, we only consider models where some symbols (both rigid and non-rigid) have a certain intended meaning. The interpretation of such *pre-defined* symbols can be fixed completely (e.g., addition) or partially (e.g., division) by means of axioms.

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 modelled by non-rigid functions. We will call these functions *program variables*. In particular, arrays and object attributes give rise to functions with arity $n > 0$ (Table 1).

We further divide program variables into heap- and stack-allocated. A variable on the heap refers to a single value and assignments changing it are immediately visible to all threads.[2] On the other hand, every thread has its own copy of each local variable (allocated on the thread's stack). An assignment to a local variable within one thread is not visible to other threads.

The local variable `v` in a concurrent program refers to a *series* of values. When the program executes, the unique value is identified by the context of the currently running thread. In the logic, we can talk about the local variable

[2] The cross-thread visibility is actually subject to conditions of the Java Memory Model. We have developed a calculus extension for checking these conditions [Kle09].

Table 1. How program variables are modelled in MODL

Program entity	modelled by	notation in logic	
local variable v (of thread t)	unary function	$v(t)$	
static field access $Class.a$	constant	$Class.a$	heap access
instance field access $o.a$	unary function	$a(o)$ or $o.a$	heap access
array access o`[`i`]`	binary function	`[]`(o,i) or o`[`i`]`	heap access

values in different threads by using a combination of variable name and thread id. All other variables (lines 2–4) are considered heap-allocated and have the same arity in programs and in the logic.

Example 3.1 (Arity of thread-local variables). Consider the concurrent MODL program `if (a>0)...` where `a` is a local variable. This thread-local variable `a` is modelled by a non-rigid function of arity 1. In the program, however, it appears without parameters, i.e, has the arity 0. Symbolic execution of this statement by a thread with id t will lead to the branch condition formula $\mathtt{a}(t) > 0$ appearing in the proof. Here the symbol `a` appears with its full arity.

Terms in MODL are defined as usual in first-order logic.

3.3. Updates

Terms and formulas can be prefixed by *updates* though. Updates can be seen as a language for describing state transitions. Evaluating $\{loc := val\}\phi$ in some state is equivalent to evaluating ϕ in a modified state where loc evaluates to val. The difference between updates and assignments is that the syntax of updates is quite restricted, making analysis and simplification of state change effects easier and efficient. Updates (together with case distinctions) can be seen as a normal form for programs and, indeed, the idea of our calculus is to stepwise transform a program to be verified into a sequence of updates, which are then simplified and applied to first-order formulas.

Definition 3.1 (Updates). The most basic update is the function update $f(t_1,\ldots,t_n) := t$, where f is a non-rigid function, and $t, t_1, \ldots, t_n$ are terms respecting the arity and typing of f.

Furthermore, if u, u_1, u_2 are updates, then the following are also updates: sequential update $u_1\,;u_2$, parallel update $u_1 \,||\, u_2$, quantified update `for` x; ϕ; u.

In both sequential and parallel[3] updates, a later sub-update to the same location overrides an earlier one. The difference however is that with sequential updates the evaluation of the second sub-update is affected by the evaluation of the first one. This is not the case for parallel updates, which are evaluated simultaneously.

Example 3.2. Consider the updates $\mathtt{x} := \mathtt{x}+1\,;\mathtt{x} := \mathtt{x}+2$ and $\mathtt{x} := \mathtt{x}+1 \,||\, \mathtt{x} := \mathtt{x}+2$ where `x` is a program variable (non-rigid constant). Evaluating these updates in a state satisfying $\mathtt{x} = 0$ results in a state satisfying $\mathtt{x} = 3$ in the first case resp. $\mathtt{x} = 2$ in the second case.

Quantified updates are a generalisation of parallel updates. A quantified update (`for` x; ϕ; u) can be understood as (the possibly infinite) sequence of updates

$$\cdots \,||\, [x/t_n]u \,||\, \cdots \,||\, [x/t_0]u$$

put in parallel in a fixed canonical order. The individual updates $[x/t_n]u, \ldots, [x/t_0]u$ are obtained by substituting the free variable x in the update u with all terms $t_n, \ldots, t_0$ such that $[x/t_i]\phi$ holds.

3.4. Syntax of Programs

First, we define sequential programs of MODL, which later serve as building blocks for concurrent programs.

Our sequential programs have several technical peculiarities[4]:

[3] Despite what their name suggests, parallel updates are not related to concurrency.

[4] The technical definitions of MODL programs do not pose restrictions on the set of Java programs that can be verified with MODL beyond those that have been stated in the introduction (⇒ Sect. 1.2).

- There is a `stop` statement, which does nothing and is never enabled. This statement is of little use in the sequential case, but is used to model concurrent programs with several thread classes.
- There is an atomic block construct, which, again, only becomes useful when the programming language is extended with concurrency.
- Every sequential program is identified with some thread executing it. All local variables are augmented with this thread id as an argument.
- Assignments must not contain more than one heap access. This restriction is necessary to faithfully model the semantics of concurrent Java assignments. We consider assignments to be atomic in our language, while they indeed can be non-atomic in Java. A program with more than one heap access in an assignment is transformed into a program satisfying the above condition by adding assignments that store the value of heap-allocated variables in fresh local variables.
- Conditions of if-then-else statements must be local variables not occurring in the then- or else-part of the statement. This restriction is similarly easy to satisfy by adding assignments with fresh local variables. The fact that these variables—once set—cannot change their value eliminates technical difficulties when specifying execution path conditions.

Definition 3.2 (Sequential programs). The set of sequential programs is recursively defined as follows. For all thread ids τ:

(Assignment statement)
$f(t_1, \ldots, t_n)$=t; is a program if

1. f is a non-rigid function symbol of arity n
2. $t_1, \ldots, t_n$, as well as t are terms correctly typed w.r.t. f
3. the assignment contains at most one heap access.

(Conditional statement)
`if (`$v(\tau)$`) {`p`} else {`q`}` is a program if p and q are programs and $v(\tau)$ is a thread-local boolean variable not appearing in p or q.

(Loop statement)
`while (`$v(\tau)$`) {`p`}` is a program if p is a program, and $v(\tau)$ is a thread-local boolean variable.

(Stop statement)
`stop` is a program.

(Sequential composition)
pq is a program if p and q are programs.

(Atomic block (also a statement))
$\ll p \gg$ is a program if p is a program.

(Lock acquisition statement)
$o(\tau)$`.<lock>();` is a program if $o(\tau)$ is a thread-local reference-valued variable.

(Lock release statement)
$o(\tau)$`.<unlock>();` is a program if $o(\tau)$ is a thread-local reference-valued variable.

Under *atomic statements* in a program, we understand the assignments, atomic blocks, lock acquisition and release statements, as well as `stop`.

Example 3.3 (Sequential program syntax). The following is an example of a concrete sequential program executed by thread `t`:

```
o(t).<lock>(); a(t)=o(t).sum; o(t).sum=a(t)+e(t); o(t).<unlock>();
```

We now use the sequential programming language to define concurrent programs. Conversely, the verification calculus breaks concurrent programs down into sequential fragments. The part of this process that builds a sequential program from a part of a concurrent one is called *sequential instantiation* (⇒ Def. 3.4).

Definition 3.3 (Concurrent programs). The set of concurrent programs is defined as follows. Every sequential program is a concurrent program under the following transformation/conditions:

- all occurrences of loops must be within atomic blocks
- atomic blocks may not be nested
- atomic blocks may not contain locking operations
- all function symbols representing local variables are stripped of thread identity (i.e., in contrast to sequential programs, the number of arguments is now one less than actual symbol arity)
- the last statement of the program must be `stop`

- `stop` may only occur on the top level in a program (not within a loop, a conditional, or an atomic block).

Example 3.4 (Concurrent program syntax). The following is an example of a concrete concurrent program with one thread class:

```
o.<lock>(); a=o.sum; o.sum=a+e; o.<unlock>(); stop; .
```

The purpose of the final `stop` statement is to provide a "parking position" for the threads that have run to completion. It also helps in writing programs with several thread classes. The following is an example of a concrete concurrent program with two thread classes:

```
x=1; stop; x=2; stop; .
```

The unfolded `run()` methods are sequentially composed, separated by `stop` statements. Since threads already execute concurrently, MODL lacks a dedicated parallel composition operator $||$.

We will omit the final `stop` statement from concurrent programs whenever clarity is not sacrificed.

Definition 3.4 (Sequential instantiation). If p is a concurrent program and τ is a thread id, then the *sequential instantiation* $p^{*(\tau)}$ is a sequential program built by augmenting every thread-local variable v in p by the thread id, giving $v(\tau)$.

We define a sequential instantiation in an analogous manner also for terms.

3.5. Program Positions, Control Variables, and Thread Configurations

Until now, we have dealt with syntactic programs, which are just templates for threads to execute. Now we introduce means to describe which threads are executing a program, and where exactly each thread is at any given moment. For this, we number all atomic statements in a program (these are: assignments, atomic blocks, lock acquisition and release statements, as well as `stop`) 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 atomic statement when it is next scheduled to run. We will refer to the statement at position i in a program p as $p(i)$ and to the total number of positions as $size(p)$.

Every position i is associated with a *control variable* $pos(i)$, which is a set-valued variable not occurring in programs. The control variable lists exactly the tids waiting to be scheduled at the resp. position. Together, the control variables specify the *thread configuration*.

Definition 3.5 (Thread configuration). A thread configuration for a program p is a non-rigid function pos^p

$$pos^p \colon \{1,\ldots,size(p)\} \to 2^{\mathscr{T}} \quad .$$

In order not to clutter notation, we will omit the program index and just write pos, as p is always clear from the context.

Example 3.5 (Thread configuration notation). In this example we assume that thread ids are integers. Then, the 3-tuple $(\{3,17,5\},\{\},\{2\})$ is an example of a configuration of size 3. A configuration of size n is compatible with programs that have n positions.

We write (compatible) pairs of thread configurations and programs by inlining the values of the control variables 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

$$\texttt{\{}_5\texttt{\}v=(x<10); if (v) \{}^{\texttt{\{}_{3,4}\texttt{\}a=x;}^{\texttt{\{}_1\texttt{\}x=a+1;\}}}\texttt{\}}^{\texttt{\{}_2\texttt{\}}} \quad .$$

On the formula level, if $\mathscr{U}$ is a sequence of updates and $\bar{c}|p$ is a program with an inlined thread configuration, the formula

$$\mathscr{U}\langle\bar{c}|p\rangle\phi$$

is shorthand for

$$\{pos(1) := c_1 \,||\, \ldots \,||\, pos(n) := c_n\}\mathscr{U}\langle p\rangle\phi \quad .$$

Note 3.1 (Disjointness of control variable values). In general, we expect to deal only with disjoint values of control variables: every thread can be at only one place at the same time. Our calculus preserves this property, but the user can still, of course, write a formula describing a state where this is not true. We dismiss such cases as pathological and leave the semantics of such formulas underspecified.

Definition 3.6 (Threads in a program). The set of threads in a program $\mathit{Tids}(p)$ is

$$\mathit{Tids}(p) = \bigcup_{i=1}^{n} \mathit{pos}(i)$$

for a program p with n atomic positions. Technically, this set is state-dependent, but our programs can neither create nor destroy threads.

3.6. The Scheduler

Definition 3.7 (Scheduler). For each program p, the *scheduler* is (modelled by) the non-rigid function (actually, constant)

$$\mathit{sched}^p :\to \mathscr{T} \cup \{\bot\} \ ,$$

which says which thread is to run next in a given state. In order not to clutter notation, we will omit the program index and just write *sched* in the future. The program to which the scheduler function refers is always clear from the context.

The interpretation of *sched* depends, in general, on program state, even though we try to minimise this dependency in our program semantics. Different models, furthermore, may interpret this function differently and, thus, have different schedulers. The value that *sched* returns must, of course, be compatible with other components of the state, i.e., the program variables and the control variables. To express this we first define what it means for a thread to be enabled.

Definition 3.8 (Statement Enabledness). We introduce a non-rigid predicate $\mathit{enabled}(s,t)$ capturing when a thread t is enabled to execute an atomic statement s within a concurrent program. We declare the predicate predefined, and its values are given by the following table:

Statement s	Enabledness condition $\mathit{enabled}(s,t)$
`stop`	*false*
assignment	*true*
atomic block	*true*
o`.<lock>()`	$o(t)$`.<lockcount>` $= 0 \vee o(t)$`.<lockedby>` $= t$
o`.<unlock>()`	*true*

Definition 3.9 (Thread Enabledness). The following non-rigid predicate captures when a thread t is enabled in a program p (we will, again, omit this program index in the future). The predicate is predefined with the following semantics:

$$\mathit{enabled}^p(t) = \begin{cases} \mathit{enabled}(s,t), & \text{if } t \in \mathit{Tids}(p), \\ \mathit{false}, & \text{otherwise,} \end{cases}$$

where s is the statement at which t is waiting to be scheduled. Per Note 3.1 there is at most one such statement. If there is none, the predicate evaluates to false.

We now state the scheduler axioms.

1. *The scheduler may only schedule existing threads.* Which threads "exist" is given by the control variables of the state for the program at hand:

$$\mathit{sched} \in \mathit{Tids}(p) \ . \tag{1}$$

2. *The scheduled thread must be enabled.* When a thread is enabled is defined in Def. 3.9. At this point the scheduler depends upon actual program variables.

$$\mathit{sched} = t \wedge t \neq \bot \to \mathit{enabled}(t) \tag{2}$$

3. *If no thread is enabled, the scheduler must return* $\bot$. This is the case when the program has terminated or entered deadlock.

$$\begin{array}{c} sched = \bot \leftrightarrow \\ \forall t.\, t \in Tids(p) \rightarrow \neg enabled(t) \end{array} \tag{3}$$

In general, this is already everything we assume about a scheduler. Fairness[5] or other scheduler properties are not built into our model. It is, however, possible to add further axioms restricting the function *sched*.

Note 3.2 (The reason for forbidding non-atomic loops). A problem for a deterministic scheduler model is the possibility that a program returns to a previously visited state (a kind of *déjà vu*). This situation may occur when a thread is executing a loop. In the real world, it would be unreasonable to expect that the scheduler run the same thread as last time in that state. A deterministic scheduler has no other option but to repeat the previous choice as the state completely determines the scheduler function. To avoid the unsoundness that is due to this discrepancy we have two options.

Option one is to forbid programs that exhibit problematic behaviour, i.e., programs with non-atomic loops. This is the option we have chosen for the moment. The key property of an atomic loop (as any atomic piece of code) is that it runs without interference, and we can collapse its intermediate states. From the scheduler's perspective an atomic loop is a single state transition. When all loops are thusly collapsed, threads never "jump back" in a configuration, and a program never passes the same thread configuration (and thus state) twice. By demanding that all loops are atomic we have eliminated déjà vus. This is not possible in the presence of non-atomic loops.

Option two (which is part of future work) is to extend the notion of state by adding yet another control variable: a *scheduling seed* σ. Symbolic execution would keep updating σ to guarantee that as long as a program is running, it never passes the same state twice. Depending on the implementation details, one could think of σ as a ghost loop counter or as a clock. We would then make the scheduling function depend on σ.

There is yet another potential reason to have an explicit seed. By using two different seeds it becomes possible to relate two different runs of the same program.

3.7. 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). Furthermore, MODL defines two concurrent and two sequential kinds of modal operators.

Definition 3.10 (Modal formulas of MODL). For each concurrent program p and every formula ϕ, $\langle p \rangle \phi$ (the concurrent "diamond" modality) and $[p]\phi$ (the concurrent "box" modality, which is a shorthand for $\neg\langle p \rangle \neg\phi$) are formulas.

If p is a sequential program and ϕ a formula, then $\langle \underline{p} \rangle \phi$ (the sequential "diamond" modality) and $[\underline{p}]\phi$ (the sequential "box" modality, which is a shorthand for $\neg\langle \underline{p} \rangle \neg\phi$) are formulas.

Intuitively, a diamond formula $\langle \underline{p} \rangle \phi$ (resp. its concurrent counterpart $\langle p \rangle \phi$) means that the program p in the diamond must terminate (resp. all threads must terminate) 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* p terminates. The formula $\psi \rightarrow [\underline{p}]\phi$ has the same meaning as the Hoare triple $\{\psi\} p \{\phi\}$.

4. Semantics of MODL

4.1. Models

The models used to interpret MODL formulas are Kripke structures $\mathscr{K} = (\mathscr{S}, \rho)$, where $\mathscr{S}$ is the set of program states and ρ is the transition relation interpreting programs.

The states $s \in \mathscr{S}$ provide interpretations of predicates and 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 $s_1, s_2 \in \mathscr{S}$ the universes of s_1 and s_2 are the same set U. We refer to U as *the* universe of $\mathscr{K}$. Rigid function symbols have

[5] It should be noted that Java itself is only "statistically fair."

the same interpretation for all states, while the interpretation of non-rigid function symbols may differ from state to state. Pre-defined symbols only admit interpretations adhering to their definitions. We assume that the set $\mathscr{S}$ of states of any Kripke structure consists of *all* first-order structures with signature Σ over some fixed universe and for some interpretation of the rigid symbols.

MODL incorporates two programming languages: the sequential and the concurrent one. The semantics of a program (of any language) is a relation between initial and final states. The semantics of a sequential program p is given by a transition relation on states $\rho_1(p) \subseteq \mathscr{S}^2$. Since sequential programs are deterministic, the relation is actually a partial function: $\rho_1(p)\colon \mathscr{S} \to \mathscr{S}$.

The semantics of a concurrent program p is given by a transition relation $\rho(p)$. Since our concurrent programs are deterministic by means of an underspecified scheduler, this relation is a partial function as well. The next section concentrates on defining the semantics relations of the concurrent and sequential programming languages in detail.

The valuation val_s of terms w.r.t. a given state s is as usual in first-order logic. The formal semantics of updates is given in [BHS07]; here we appeal to the informal description given in Section 3.3. The semantics of modal formulas (the validity relation $\models$) is given in the definition below, otherwise the semantics of formulas is as usual in first-order logic.

Definition 4.1 (Semantics of modal formulas). (Modalities with concurrent programs)
$s \models \langle p \rangle \phi$ iff $(s,s') \in \rho(p)$ for some state s' with $s' \models \phi$.

(Modalities with sequential programs)
$s \models \langle \underline{q} \rangle \phi$ iff $(s,s') \in \rho_1(q)$ for some state s' with $s' \models \phi$.

We say that a Kripke structure is a *model* of a formula ϕ iff $s \models \phi$ is true in all states s of that structure. A formula ϕ is *valid* if all Kripke structures are a model of ϕ.

Definition 4.2 (State variation). If $s \in \mathscr{S}$ is a state and u is an update, then $s' = s[\![u]\!]$ is a *state variation* (i.e., also a state). This means that s' is identical to s except for the interpretation mapping, which is changed according to the update u.

4.2. Semantics of Sequential Programs

Definition 4.3 (Semantics of sequential programs). The semantics of sequential programs $\rho_1(p)$ is the smallest relation satisfying the following conditions. It does not depend on the scheduler.

(Stop)
$\rho_1(\texttt{stop}) = id$

(Atomic block)
$\rho_1(\ll p \gg) = \rho_1(p)$

(Assignment)
$(s,s') \in \rho_1(f(t_1,\ldots,t_n) = t)$ iff the statement $f(t_1,\ldots,t_n) := t$ interpreted as a Java assignment does not throw an exception[6] and $s' = s[\![f(t_1,\ldots,t_n) := t]\!]$.

(Sequential composition)
$(s,s') \in \rho_1(pq)$ iff $(s,s'') \in \rho_1(p)$ and $(s'',s') \in \rho_1(q)$ for some state s''.

(Conditional)
$(s,s') \in \rho_1(\texttt{if (}v(t)\texttt{) \{}p\texttt{\} else \{}q\texttt{\}})$ iff either

(1) $val_s(v(t)) = \mathit{TRUE}$ and$(s,s') \in \rho_1(p)$, or

(2) $val_s(v(t)) = \mathit{FALSE}$ and $(s,s') \in \rho_1(q)$.

(Loop)
$(s,s') \in \rho_1(\texttt{while (}v(t)\texttt{) \{}p\texttt{\}})$ iff there is an $n \in \mathbb{N}$ and there are states $s_0,\ldots,s_n$ with $s = s_0$ and $s' = s_n$ such that

[6] We do not give a formal definition of when an assignment throws an exception, since it would require formalising here large portions of the Java Language Specification. The sequential KeY calculus does exactly this though, and, in our implementation, we delegate to it the check if an assignment succeeds.

(1) for $0 \leq i < n$, $val_{s_i}(v(t)) = TRUE$ and $(s_i, s_{i+1}) \in \rho_1(p)$, and

(2) $val_{s_n}(v(t)) = FALSE$.

(Lock acquisition)
$(s, s') \in \rho_1(o(t)\texttt{.<lock>()})$ iff either

(Case 1: the lock is free)

$$val_s(o(t)\texttt{.<lockcount>}) = 0 \text{ and } val_s(o(t)\texttt{.<lockedby>}) = \perp$$

or

$$val_s(o(t)\texttt{.<lockcount>}) > 0 \text{ and } val_s(o(t)\texttt{.<lockedby>}) = val_s(t)$$

and, in either case,

$$s' = s\left[\!\!\left[\begin{array}{c} o(t)\texttt{.<lockcount>} := o(t)\texttt{.<lockcount>} + 1 \,||\\ o(t)\texttt{.<lockedby>} := t \end{array}\right]\!\!\right]$$

or

(Case 2: the lock is taken)
$val_s(o(t)\texttt{.<lockcount>}) > 0$ and $val_s(o(t)\texttt{.<lockedby>}) \neq val_s(t)$ with $s' = s$.

(Lock release)
$(s, s') \in \rho_1(o(t)\texttt{.<unlock>()})$ iff either

(Case 1: lock depth not yet exhausted)

$$val_s(o(t)\texttt{.<lockcount>}) > 1$$
$$val_s(o(t)\texttt{.<lockedby>}) = val_s(t)$$
$$s' = s\left[\!\!\left[\begin{array}{l} o(t)\texttt{.<lockcount>} := \\ o(t)\texttt{.<lockcount>} - 1 \end{array}\right]\!\!\right]$$

or

(Case 2: lock depth exhausted)

$$val_s(o(t)\texttt{.<lockcount>}) = 1$$
$$val_s(o(t)\texttt{.<lockedby>}) = val_s(t)$$
$$s' = s\left[\!\!\left[\begin{array}{l} o(t)\texttt{.<lockcount>} := 0 \,|| \\ o(t)\texttt{.<lockedby>} := \perp \end{array}\right]\!\!\right]$$

4.3. Semantics of Concurrent Programs

To make specifying the semantics of if-statements easier we assume that every thread steps through both the then- and the else-part of all if-statements. Yet the thread can only change the state if it is in the "right" part and executes NOPs otherwise. The *path condition* tells us if we are in the right part.

Definition 4.4 (Path condition of a position in program). Let k be a position of an atomic sub-program in a non-atomic program p. Let this position occur within the scope of $n \geq 0$ (nested) if-statements in their then- or else part. Let $v_1, \ldots, v_n$ be the conditions of these if-statements.

Since, by definition, the local variable v_i does not occur in the then- or else-part of the ith if-statement, its value is not changed during the execution of the if-statement after it has been evaluated.

We define the *path condition* of position k in program p as the formula:

$$path(k, p, tid) = B_1 \wedge \ldots \wedge B_n \ ,$$

where

$$B_i = \begin{cases} \big(v_i(tid) = TRUE\big), & \text{if } k \text{ is in the then-part of the } i\text{th if-statement} \\ \big(v_i(tid) = FALSE\big), & \text{if } k \text{ is in the else-part.} \end{cases}$$

Thus, a thread t will execute the atomic program at k within p iff $path(k, p, t)$ holds.

Example 4.1. The path condition of the statement `l=r;` in the program

```
if (a) {if (b) {} else {l=r;}} else {}
```

for a thread t is

$\mathtt{a}(t) = \mathit{TRUE} \wedge \mathtt{b}(t) = \mathit{FALSE}$.

Our next goal is to define the semantics of concurrent programs $\rho(\cdot)$. The base for this is the semantics of sequential programs $\rho_1(\cdot)$. We use $\rho_1(\cdot)$ to describe the first step in the execution of a concurrent program, as scheduled by the scheduler function. All further steps of the concurrent program are handled by recursively repeating the process.

Definition 4.5 (Semantics of concurrent programs). The semantics $\rho(p)$ of a concurrent program p is inductively defined as the smallest relation such that:

- $(s,s) \in \rho(p)$ if no thread of p is enabled in s, i.e., $\mathit{sched} = \bot$ in s.
- $(s,s') \in \rho(p)$ if some thread of p is enabled in s , and

  (1) $\mathit{sched} = \mathit{tid}$ in s

  (2) $\mathit{tid} \in \mathit{pos}(i)$ in s (we assume there is always exactly one such i, cf. Note 3.1)

  (3) q is the atomic sub-program at position i in p

  (4) $s \models \mathit{path}(\mathrm{i}, p, \mathit{tid})$,

  (5) $(s,s'') \in \rho_1(q^{*(\mathit{tid})})$ for a state s''

  (6) $(s''[\![\mathit{pos}(i) := \mathit{pos}(i) \setminus \{\mathit{tid}\} \,||\, \mathit{pos}(i+1) := \mathit{pos}(i+1) \cup \{\mathit{tid}\}]\!], s') \in \rho(p)$
- $(s,s') \in \rho(p)$ if some thread of p is enabled in s , and

  (1)-(3) as above

  (4) $s \not\models \mathit{path}(i, p, \mathit{tid})$,

  (5) there is a state $s'' = s[\![\mathit{pos}(i) := \mathit{pos}(i) \setminus \{\mathit{tid}\} \,||\, \mathit{pos}(i+1) := \mathit{pos}(i+1) \cup \{\mathit{tid}\}]\!]$

  (6) $(s'', s') \in \rho(p)$

5. A More Verification-Friendly Version of MODL

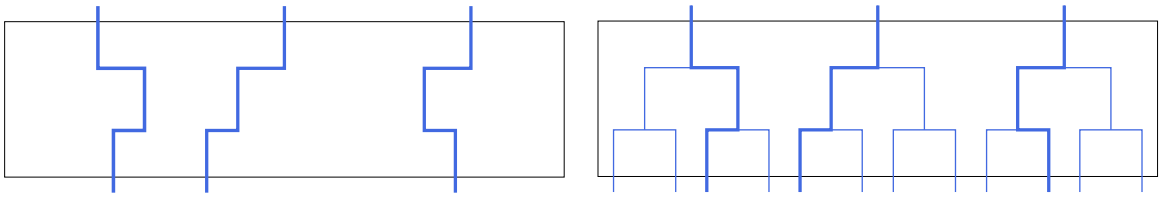
The logic presented so far already gives a complete account of multi-threading for the chosen language fragment and even allows symbolic execution of programs. It has two deficiencies though:

- The threads involved have to be explicitly enumerated. Configurations are sets of thread ids, even though the concrete ids are actually not important. This circumstance makes it impossible to make statements about an unbounded (fixed but unknown) number of threads.
- Transitions are always totally ordered (resulting in proof branching), even if they are independent. Consider two threads τ_1 and τ_2 that are ready to be scheduled at the same position. We have no choice but to perform a case distinction which one will run first, even if this choice is not important. Since we are dealing with symbolic data, this distinction is seldom important. Nonetheless, we have to choose, which prevents us from verifying programs with an unbounded number of threads.

To overcome these deficiencies, we have developed a more refined logic where configurations are not enumerated, but described algebraically. Efficient laws for reasoning about these descriptions complete the picture. The basis for the efficiency gain is symbolic thread symmetry.

5.1. 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, for instance, in [RDHI03] (the Bogor tool) and [Yah01] (on-the-fly model-checking with TVLA). Due to their nature, these approaches only apply symmetry reduction to threads with exactly the same concrete local data (inapplicable in Figure 2a). In a deductive verification system we can give this idea a new twist.



(a) concrete data (b) symbolic data

Fig. 2. Explored thread trajectories in a program

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 behaviour in terms of their execution path through the code. The number of different paths is, furthermore, finite and relatively small; it is bounded by the shape of the program text.[7]

Since we are executing programs symbolically, we are already exploring more paths than indicated by any concrete execution. Having paid the price in case distinctions, we might as well reap the benefits and identify threads with different local data (Figure 2b). With this technique, we can in many cases eliminate the necessity of considering 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 concurrent systems feasible.

5.2. Expressing Unbounded Concurrency

We "force" each thread to linearly traverse the program: There is no jumping back (except within an atomic loop, cf. Note 3.2), and each thread visits each position at most once (never, if it gets stuck along the way in an atomic loop, or upon lock acquisition).[8] Assuming threads with tids $1,\dots,n$, it is clear that for every position i, there is a permutation $\pi_i : \{1\dots n\} \to \{1\dots n\}$, which describes the order in which the threads are scheduled at this position (should they reach it).

Given these permutations, it is sufficient to know *how many* threads are at each position, to fix the exact thread configuration as defined in Def. 3.5. Configurations with m positions can be now written as $(\pi_0, \pi_1{:}k_1, \dots, \pi_{m-1}{:}k_{m-1}, k_m)$, where $\pi_0,\dots,\pi_{m-1}$ are terms representing the permutations and $k_1,\dots,k_m$ are terms representing the number of threads. We give the details of how the two notations are related in the following.

5.3. Describing Thread Configurations

Definition 5.1 (Thread configuration). Configurations with explicit tids were introduced in Def. 3.5. We now overload this term with the following definition. Unless explicitly stated otherwise, in the following, all configurations refer to this formulation.

A thread configuration for a program p is a family of non-rigid functions

$$\pi_i^p : \mathbb{N} \to \mathscr{T} \quad \text{for } i \in \{0,\dots,size(p)-1\}$$

together with a non-rigid function

$$pos^p : \{1,\dots,size(p)\} \to \mathbb{N} \ .$$

π_i is a permutation of the set of tids $\mathscr{T}$, encapsulating the scheduler decisions at position i.[9] $pos(i)$ is the number of threads currently available for scheduling at position i.

In order not to clutter notation, we will omit the program index and just write π_i and pos. The program they refer to is always clear from the context. As before, we also often present configurations as inlined within programs. This time

[7] Remember that we only consider atomic loops, which can be compressed into a single (complex) computation step.

[8] This does mean that threads can end up in "wrong" parts of if-then-else statements. To preserve the original semantics of the program, we define that the state is not changed by the program while its control flow is in the "wrong" part.

[9] The "ghost position" 0 will be explained later.

we limit ourselves to the values of *pos*. Since we never deal with concrete values of π_i, we omit them when stating configurations and simply imply their existence.

Example 5.1. Consider a program of size four with 2, 3, 5 and 7 threads waiting at each position respectively. The thread configuration of this program consists of the non-rigid function *pos* (with $pos(1) = 2$, $pos(2) = 3$, $pos(3) = 5$, $pos(4) = 7$), and the four non-rigid "permutation" functions $\pi_0, \ldots, \pi_3$ (whose values we do not know). Altogether there are 17 threads.

If we concentrate on position 1, we can see that $3+5+7 = 15$ threads have already passed this position and the next one to execute will be the 16th in count. If we now concentrate on position 2, we can see that $5+7 = 12$ threads have already passed this position and the next one to execute will be the 13th in count.

Definition 5.2 (*Post*$(\cdot)$). For a given (implied) program p and a position $i \leq size(p)$, we define a predefined non-rigid function $Post(i)$ with the semantics fixed by:

$$Post(i) = \begin{cases} pos(i) & \text{if } i = size(p), \text{ or if the statement at position } i \text{ in } p \text{ is } \texttt{stop} \\ pos(i) + Post(i+1) & \text{otherwise.} \end{cases}$$

$Post(i)$ is the number of threads in p that have already passed position i in the current state; though only threads in the thread class associated with i are counted. If there is only one thread class, the situation is simpler:

$$Post(i) = pos(i+1) + \ldots + pos(size(p)) \ .$$

Example 5.2 (Example 5.1 continued). So, in our example $Post(2) = 5+7 = 12$. The next thread scheduled at position 2 is the $(Post(2)+1) = 13$th thread. But exactly which one is the 13th? Here the permutation functions come into play. The exact tid of the thread scheduled to run next at position 2 is given by $\pi_2(Post(2)+1) = \pi_2(13)$. This way we can talk concisely about thread orderings even if we do not 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 are mostly agnostic of scheduler decisions, as they are 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.

As mentioned above, the *pos* and the π_i functions completely fix the thread lineup. We now state exactly how, by defining the function pos_γ which in any given state produces an enumerative configuration in the sense of Def. 3.5.

Definition 5.3 (Configuration concretisation). A *concretisation function* (of size n) is a predefined non-rigid function

$$pos_\gamma \colon \{1, \ldots, n\} \to 2^{\mathscr{T}}$$

with the semantics fixed by

$$pos_\gamma(i) = \Big\{ \pi_{i-1}(1), \ldots, \pi_{i-1}(Post(i-1)) \Big\} \setminus \Big\{ \pi_i(1), \ldots, \pi_i(Post(i)) \Big\} \ .$$

If i is the last position in a thread class, then the subtrahend (second term) is omitted.

The intuition behind this definition is the following. The threads waiting at position i are exactly those that have already passed the position $i-1$, but excluding those that have already moved on past i.

At this point, the necessity for a ghost position number zero becomes apparent. While we never need to know the value of $pos(0)$, the permutation function π_0 is needed to give identity to the threads waiting at position one. In some sense, it provides for canonical ids for all threads in a configuration, regardless of their position. In Example 5.1, the seventeen threads present in the system draw their ids from the set $\{\pi_0(1), \ldots, \pi_0(17)\}$.

Example 5.3 (Example 5.1 continued). We now translate the four integers and the four permutations from above

into an enumerative 4-set configuration:

$$\begin{pmatrix} pos_\gamma(1), \\ pos_\gamma(2), \\ pos_\gamma(3), \\ pos_\gamma(4) \end{pmatrix} = \begin{pmatrix} \{\pi_0(1),\ldots,\pi_0(17)\} \setminus \{\pi_1(1),\ldots,\pi_1(15)\}, \\ \{\pi_1(1),\ldots,\pi_1(15)\} \setminus \{\pi_2(1),\ldots,\pi_2(12)\}, \\ \{\pi_2(1),\ldots,\pi_2(12)\} \setminus \{\pi_3(1),\ldots,\pi_3(7)\}, \\ \{\pi_3(1),\ldots,\pi_3(7)\} \end{pmatrix}$$

5.4. New Scheduler Formalisation

Since we are aiming towards identifying all threads that have reached a certain position within the program, we wish to decompose the scheduling function into two components: the position choice function $\mathscr{P}$ and the thread choice functions π_i. In the following we will be restating the important definitions of concrete MODL primarily in terms of positions instead of in terms of threads.

The main component of the new scheduler formalisation is the *position choice* function $\mathscr{P}$. It returns the position from which the next thread will be scheduled in the current state—or $\bot$, if no enabled positions ($\Rightarrow$ Def. 5.4) remain.

Putting $\mathscr{P}$ together with the permutations introduced in the previous section, we obtain the following decomposition of the scheduler function (for non-disabled configurations):

$$sched = \pi_{\mathscr{P}}(Post(\mathscr{P}) + 1) \ . \tag{4}$$

Position choice function characterisation In this section we state axioms about the position choice function, but first we need to define when a position is enabled. Informally, position i is *enabled* in a configuration iff its tid set is not empty and its statement is enabled ($\Rightarrow$ Def. 3.8) for some thread at this position.

Definition 5.4 (Position enabledness). We introduce a non-rigid predicate $enabled^p(i)$ capturing when a position i is enabled in a program p (which we will omit, as it is clear from the context). We declare the predicate predefined, and its semantics is given by the following equality:

$$enabled(i) = \exists t. \left(t \in pos_\gamma(i) \wedge \big(path(i,p,t) \rightarrow enabled(p(i),t)\big) \right) \ .$$

Note 5.1. In many cases the existential quantifier can be trivially eliminated. A `stop` statement is never enabled, while for assignments, atomic blocks, and lock releases, the statement is always enabled ($\Rightarrow$ Def. 3.8), and the above equality simplifies to

$$enabled(i) = \exists t. t \in pos_\gamma(i) \ ,$$

which means nothing else than

$$enabled(i) = pos(i) > 0 \ .$$

A similar simplification applies when all threads compete for the same lock o (assuming absence of reentrant locking and the same path condition). Then $enabled(i)$ becomes

$$pos(i) > 0 \wedge o.\texttt{<lockcount>} = 0 \ .$$

These are exactly the cases of full symmetry between threads.

Having defined position enabledness, we now axiomatise the position choice function. To achieve an adequate scheduler representation, the position choice function is subject to the following axioms:

- Only valid positions (or $\bot$) are returned:

  $$\mathscr{P} = \bot \vee 1 \leq \mathscr{P} < size(p). \tag{5}$$

  This axiom effectively amounts to a disjunction over the positions of p, which during the proof gives rise to a case distinction. Note that $size(p)$ is never returned, since the last position must be a `stop`, which is never enabled.

- The non-$\bot$ values of $\mathscr{P}$ are further restricted to the positions enabled in a given configuration:

$$\mathscr{P} \neq \bot \rightarrow \mathit{enabled}(\mathscr{P}) \ . \tag{6}$$

- $\mathscr{P}$ may only return $\bot$ if no position is enabled:

$$\begin{array}{l} \mathscr{P} = \bot \rightarrow \\ \forall i. \big(1 \leq i < \mathit{size}(p) \rightarrow \neg \mathit{enabled}(i)\big) \ . \end{array} \tag{7}$$

Thread choice function characterisation Each thread choice function π_i is in every state an injective mapping from $\mathbb{N}$ to the set of tids $\mathscr{T}$ (we assume there are infinitely many thread ids). The injectivity is based on the fact that no thread can pass the same position twice as we have ruled out non-atomic loops. Formally:

$$\pi_k(i) = \pi_k(j) \text{ iff } i = j \ . \tag{8}$$

While it is our goal to assume as little about the thread choice functions as possible, in reality they are not completely arbitrary. The thread choice function at position i can only choose threads available at this position.

$$\pi_i(\mathit{Post}(i)+1) \in \mathit{pos}_\gamma(i) \tag{9}$$

This constraint ties the choice at position i to the choices made at previous positions. Our calculus uses the axioms presented here to gather a constraint on the scheduler as it explores the behaviour of the program.

Note 5.2. For efficient reasoning, the formula $\pi_i(\mathit{Post}(i)+1) \in \mathit{pos}_\gamma(i)$, which is shorthand for

$$\pi_i(\mathit{Post}(i)+1) \in \Big\{ \pi_{i-1}(1), \ldots, \pi_{i-1}(\mathit{Post}(i-1)) \Big\} \setminus \Big\{ \pi_i(1), \ldots, \pi_i(\mathit{Post}(i)) \Big\} \ ,$$

can be simplified. The subtracted term can be safely dropped if we recall injectivity of π_i. Together with (8) it is sufficient to demand that:

$$\pi_i(\mathit{Post}(i)+1) \in \Big\{ \pi_{i-1}(1), \ldots, \pi_{i-1}(\mathit{Post}(i-1)) \Big\} \ .$$

Finally, the threads of different thread classes are never confused. If there is a `stop` statement at position b in a program, then

$$\forall i, j, x, y. \ \big(i < b \wedge j \geq b \ \rightarrow \pi_i(x) \neq \pi_j(y)\big) \ .$$

New definition of program semantics In parallel to Def. 4.5, we now state a new definition of program semantics.

Definition 5.5 (Semantics of concurrent programs). The semantics $\rho(p)$ of a concurrent program p is inductively defined as the smallest relation such that:

- $(s,s) \in \rho(p)$ if no position of p is enabled in s, i.e., $\mathscr{P} = \bot$ in s.
- $(s,s') \in \rho(p)$ if some position of p is enabled in s , and

  (1) $\mathit{tid} = \pi_{\mathscr{P}}(\mathit{Post}(\mathscr{P})+1)$ in s

  (2) q is the atomic sub-program at position $\mathscr{P}$ in p

  (3) $s \models \mathit{path}(\mathscr{P}, p, \mathit{tid})$,

  (4) $(s,s'') \in \rho_1(q^{*(\mathit{tid})})$ for a state s''

  (5) $(s''[\![\mathit{pos}(\mathscr{P}) := \mathit{pos}(\mathscr{P}) - 1 \,||\, \mathit{pos}(\mathscr{P}+1) := \mathit{pos}(\mathscr{P}+1)+1]\!], s') \in \rho(p)$
- $(s,s') \in \rho(p)$ if some position of p is enabled in s , and

  (1)-(2) as above

  (3) $s \not\models \mathit{path}(\mathscr{P}, p, \mathit{tid})$,

  (4) there is a state $s'' = s[\![\mathit{pos}(\mathscr{P}) := \mathit{pos}(\mathscr{P}) - 1 \,||\, \mathit{pos}(\mathscr{P}+1) := \mathit{pos}(\mathscr{P}+1)+1]\!]$

  (5) $(s'', s') \in \rho(p)$

6. A Calculus for MODL

6.1. Calculus Overview

To prove formulas of MODL, we developed a sequent calculus. A *sequent* is of the form $\Gamma \Longrightarrow \Delta$, where Γ and Δ are sets of formulas. Its informal semantics is the same as that of the formula $\bigwedge_{\phi\in\Gamma}\phi \;\rightarrow\; \bigvee_{\psi\in\Delta}\psi$.

A *rule schema* is of the form

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_k}{C} \qquad (k \geq 0)$$

where $P_1,\ldots,P_k$ and C are schematic sequents, i.e., sequents containing schema variables. 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 6.6 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 $\mathscr{U}$, 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.

The calculus is built from the following newly-developed components:

A. rules for symbolic execution of concurrent programs (interleaving and symmetry reduction) ($\Rightarrow$ Sect. 6.3)
B. rules for reasoning about scheduling functions (permutations) produced by Component A (presented as axioms in Sect. 5.4)
C. concurrent invariant rule (not needed for completeness) ($\Rightarrow$ Sect. 6.6)
D. unfolding rules for translating Java to MODL ($\Rightarrow$ Sect. 6.2)

as well as the pre-existing components of the sequential KeY calculus:

1. FOL rules, reasoning about equality and arithmetic, induction
2. rules for symbolic execution of atomic sequential program fragments produced by Component A
3. invariant rule for sequential loops
4. method contract rules (further modularisation)
5. rules for simplification and application of updates, which are produced by Component 2 (efficient aliasing treatment)

The Components 1–5 have been borrowed (with very minor modifications) from the stock KeY system.

6.2. Program Unfolding: Translating Java to MODL

Our calculus is designed for verification of MODL programs. In the following we sketch a mapping from Java programs satisfying the requirements given in the introduction to MODL. The mapping is such that the Java program and its counterpart in MODL perform the same state transition: if started in the same state both will either terminate in the same state or not terminate at all. Thus, if the MODL program can be verified, then the original Java program is correct as well.

The mapping is such that the Java program and the result in MODL have the same input/output relation. If the MODL program can be verified, then the same can be assumed about the Java program.

Translating Java to MODL is a two-step process. First, we completely *unfold* the Java program using a special rule set in KeY. The result is a more fine-grained Java program that is semantically equivalent to the original. Then, we use a simple transformation from the unfolded program into MODL.

Unfolding the Java program For a concurrent Java program α, the unfolded Java program α' satisfies the following conditions:

1. α' is trace-equivalent to α (w.r.t. vocabulary of α)
2. all occurring expressions are in normal form, i.e., it is no longer possible to factor out subexpressions by means of fresh local variables
3. each assignment is atomic (i.e., updates at most one heap location)

Table 2. Examples of unfolding Java programs

Java statement	unfolded form
`o.a=u.a++;`	`v=u.a; u.a=v+1; o.a=v;`
`if (o.a>1) {`α`} else {`β`}`	`v=o.a>1; if (v) {`α'`} else {`β'`}`
`while (o.a>1) {`α`}`	`v=o.a>1; while (v) {`α' `v=o.a>1;}`
`synchronized(`o`) { `α `}`[†]	o`.<lock>(); `α' o`.<unlock>();`

v is in each case a fresh local variable of appropriate type

[†] The correct way to unfold a synchronized block is actually `try {`o`.<lock>(); `α`} finally {`o`.<unlock>();}`, but since we do not allow catching exceptions at the moment, we are using a simpler version.

$$\text{step}\ \frac{\begin{array}{rl} & \Longrightarrow \mathscr{P} = i \\ path(i,p,tid) & \Longrightarrow \langle\!\![\underline{S^{*(tid)}}]\!\!\rangle \langle\!\![\alpha^{\{n-1\}} S^{\{k+1\}} \omega]\!\!\rangle \phi \\ \neg path(i,p,tid) & \Longrightarrow \langle\!\![\alpha^{\{n-1\}} S^{\{k+1\}} \omega]\!\!\rangle \phi \end{array}}{\begin{array}{c} \Longrightarrow \langle\!\![\alpha^{\{n\}} S^{\{k\}} \omega]\!\!\rangle \phi \\ \uparrow \\ \text{position } i \text{ in } p \end{array}}$$

Fig. 3. The concurrent symbolic execution rule

4. the conditions of if-statements and loops are fresh local variables
5. the conditions of if-statements do not occur in the then- or else-part of the statement
6. method calls are inlined, if necessary together with extra conditionals to simulate dynamic binding.

Examples of concurrent program unfolding are given in Table 2. To achieve this effect, we utilise the rules that are already part of the sequential KeY calculus. These rules introduce fresh local variables and additional assignments. Furthermore, instance creation is already modelled by assignment to ghost fields in the KeY calculus, and method implementations are inlined.

We have manually inspected the KeY rule base identifying the rules that perform unfolding. Syntactically, this set of rules can be very closely approximated as a set of rules that match programs, but do not produce updates or case distinctions. Minor fine-tuning was subsequently performed to ensure the atomicity condition of assignments. We have also checked that no rules "swallow" intermediate states, i.e., perform optimisations like replacing `i++;i--;` by a NOP. The resulting rules were then pooled in a special *unfolding strategy* of the prover.

Translating unfolded Java into MODL After the program has been completely unfolded, it almost satisfies the syntax requirements of MODL. The biggest missing piece is the atomicity requirement for loops. The user must declare code sections containing loops as atomic. More atomic sections can be introduced in order to improve proof performance. In both cases, one needs to carry out further justification (see Section 8.3 for an example). Finally, it remains to compose different thread classes by means of `stop` statements, add initial thread configurations and, in general, formulate the proof obligation.

6.3. The Basic Rules of Concurrent Execution

The calculus presented in the following makes extensive use of the axioms given previously. The axioms are the constraints on the interpretation of predefined functions and predicates given in their definitions. These axioms can be added to the antecedent of a proof goal at any time. Among symbols subject to axioms are enabledness predicates (⇒ Def. 5.4), path conditions (⇒ Def. 4.4), scheduler functions (⇒ Sect. 5.4), etc.

Figure 3 shows the main rule of MODL calculus. The rule shows how to symbolically execute any atomic statement that is not a lock acquisition or release. In the rule, α and ω denote unchanged program parts, and i is the position of the executed atomic statement S (in the overall program p). $S^{*(tid)}$ is the sequential instantiation (⇒ Def. 3.4) of S for the currently running thread *tid*, which is an abbreviation for:

$$tid = \pi_i(Post(i)+1) \ .$$

The formula $path(i,p,tid)$ is the path condition (⇒ Def. 4.4) of the statement S in p for thread tid. $\mathscr{P}$ is the position choice function, and the first premiss encodes the scheduler decision to schedule a thread at position i next. Since the scheduler behaviour is in general unknown, this rule is usually applied after a case distinction over possible values of $\mathscr{P}$. These are, in turn, dictated by the scheduler axioms (⇒ Sect. 5.4).

After applying the step rule, the sequential program $S^{*(tid)}$ has to be tackled by the rules of the sequential KeY calculus. Eventually, it will be reduced to a series of updates and case distinctions.

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

$$\text{empty-program}\ \frac{\Longrightarrow \mathscr{P} = \bot \qquad \Longrightarrow \phi}{\Longrightarrow \langle [p] \rangle \phi}$$

6.4. A Simple but Complete Verification Example

The following example is popular in the field (e.g., [AFF06]), since it already exhibits a large part of the issues inherent to thread-based concurrency.

Example 6.1. 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 static variable and `e` is a thread-local variable containing the payment amount.

$$\forall n.\ \{\texttt{sum} := 0\}\langle^{\{n\}} \ll \texttt{sum=sum+e;} \gg^{\{\}} \rangle (\texttt{sum} = \sum_{i=1}^{n} \texttt{e}(\pi_0(i))) \tag{10}$$

Note that for presentation purposes we have abused the programming language by writing an assignment with two heap accesses. This shorthand is permissible here, since the assignment is protected by an atomic block. This protection ensures that the assignment `a=sum+e;sum=a;` (as the above is properly written) does not lead to an atomicity failure (sometimes known as "race").

As the first step of the proof, we eliminate the universal quantifier from the conjecture, replacing n by a Skolem constant n_0. Then we apply induction. The induction hypothesis is that $n_0 - k$ transactions have been completed, while k remain (k is the induction variable, $0 \leq k \leq n_0$):

$$\{\texttt{sum} := \sum_{i=1}^{n_0-k} \texttt{e}(\pi_1(i))\}\langle^{\{k\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0-k\}} \rangle (\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

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

$$\{\texttt{sum} := \sum_{i=1}^{n_0-k-1} \texttt{e}(\pi_1(i))\}\langle^{\{k+1\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0-k-1\}} \rangle (\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

We apply the step rule once. There is only one position and thus one relevant permutation, namely π_1. The position is enabled (as $k+1 > 0$), and there is indeed only one possible choice $\mathscr{P} = 1$ (per Axioms (5) and (6) on page 17). Since there are no if-statements, the path condition is simply *true*. The only remaining goal is thus:

$$\{\texttt{sum} := \sum_{i=1}^{n_0-k-1} \texttt{e}(\pi_1(i))\}\langle \underline{\texttt{sum=sum+e;}}^{*(\pi_1(n_0-k))} \rangle \langle^{\{k\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0-k\}} \rangle (\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

We expand the definition of sequential instantiation. Only the thread-local variable `e` is affected:

$$\{\texttt{sum} := \sum_{i=1}^{n_0-k-1} \texttt{e}(\pi_1(i))\}\langle \underline{\texttt{sum=sum+e(}\pi_1(n_0-k)\texttt{);}} \rangle \langle^{\{k\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0-k\}} \rangle (\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

We execute the sequential instantiation of the assignment symbolically using the sequential assignment rule. This generates the update $\{\texttt{sum} := \texttt{sum} + \texttt{e}(\pi_1(n_0-k))\}$. We have:

$$\{\texttt{sum} := \sum_{i=1}^{n_0-k-1} \texttt{e}(\pi_1(i))\}\{\texttt{sum} := \texttt{sum} + \texttt{e}(\pi_1(n_0-k))\}\langle^{\{k\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0-k\}} \rangle (\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

Update simplification yields:

$$\{\texttt{sum} := \sum_{i=1}^{n_0-k} \texttt{e}(\pi_1(i))\}\langle^{\{k\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0-k\}}\rangle(\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

Now, the induction hypothesis for k applies, and the step case of the induction is closed.

Base case The base case $k = 0$ looks like this:

$$\{\texttt{sum} := \sum_{i=1}^{n_0} \texttt{e}(\pi_1(i))\}\langle^{\{0\}} \ll \texttt{sum=sum+e;} \gg^{\{n_0\}}\rangle(\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

There are no enabled threads left, so the modality with the program can be removed (rule empty-program), leaving to prove:

$$\{pos(1) := 0 \,||\, pos(2) := n_0\}\{\texttt{sum} := \sum_{i=1}^{n_0} \texttt{e}(\pi_1(i))\}(\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

After applying the inner update, the goal is:

$$\{pos(1) := 0 \,||\, pos(2) := n_0\}(\sum_{i=1}^{n_0} \texttt{e}(\pi_1(i)) = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

The equality of the two sums follows from commutativity of addition, the injectivity of π_i (Axiom 8), and the fact that $\{\pi_0(1),\ldots,\pi_0(n_0)\} = \{\pi_1(1),\ldots,\pi_1(n_0)\}$, which, in turn, follows from the definition of position concretisation for position 1 ($\Rightarrow$ Def. 5.3):

$$pos_\gamma(1) = \left\{\pi_0(1),\ldots,\pi_0(n_0)\right\} \setminus \left\{\pi_1(1),\ldots,\pi_1(n_0)\right\} \ .$$

Taking into account that $pos_\gamma(1) = \emptyset$ (as $pos(1) = 0$), we obtain the desired set equality:

$$\left\{\pi_0(1),\ldots,\pi_0(n_0)\right\} = \left\{\pi_1(1),\ldots,\pi_1(n_0)\right\} \ .$$

This completes the base case proof.

Use case By this argument we have established the hypothesis for any $k \leq n_0$. Instantiating k with n_0 yields:

$$\{\texttt{sum} := \sum_{i=1}^{0} \texttt{e}(\pi_1(i))\}\langle^{\{n_0\}} \ll \texttt{sum=sum+e;} \gg^{\{0\}}\rangle(\texttt{sum} = \sum_{i=1}^{n_0} \texttt{e}(\pi_0(i)))$$

The sum in the update collapses yielding the Skolemised version of the original conjecture (10).

The lessons learned from the example are: We have verified the transaction mechanism for an arbitrary number of threads. This is important, since it is 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. The scheduling-independence of the system does not require a separate proof before the functional properties can be addressed. Furthermore, it is possible to apply the full power of deductive reasoning about unbounded data and its implementations (e.g., overflow control for the integer variables [BS05]).

6.5. Treating Locking

The lock acquisition method is symbolically executed by applying the rule shown in Figure 4. 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, $\mathscr{P} = i$ implies $enabled(i)$). As before, the thread performing the acquire has the id $tid = \pi_i(Post(i)+1)$.

Note that the mutual-exclusion semantics of locking does not appear in the rule *directly*. Rather, it is hidden in the definition of enabledness ($\Rightarrow$ Def. 5.4, 3.8), which in its turn is part of the axiomatisation of position choice $\mathscr{P}$.

$$\Longrightarrow \mathscr{P} = i$$

$$path(i,p,tid) \Longrightarrow \{o^{*(tid)}.\texttt{<lockcount>} := o^{*(tid)}.\texttt{<lockcount>} + 1\}\ \{o^{*(tid)}.\texttt{<lockedby>} := tid\}\ \langle\!|\alpha\ \ {}^{\{n-1\}}o.\texttt{<lock>}()^{\{k+1\}}\ \ \omega|\!\rangle\phi$$

$$\text{lock}\ \frac{\neg path(i,p,tid) \Longrightarrow \langle\!|\alpha\ \ {}^{\{n-1\}}o.\texttt{<lock>}()^{\{k+1\}}\ \ \omega|\!\rangle\phi}{\Longrightarrow \langle\!|\alpha\ \ \underbrace{{}^{\{n\}}o.\texttt{<lock>}()^{\{k\}}}_{\text{at position } i \text{ in } p}\ \ \omega|\!\rangle\phi}$$

Fig. 4. The rule for lock acquisition

$$\Longrightarrow \mathscr{P} = i$$

$$path(i,p,tid) \Longrightarrow \{o^{*(tid)}.\texttt{<lockcount>} := o^{*(tid)}.\texttt{<lockcount>} - 1\}\ \langle\!|\alpha\ \ {}^{\{n-1\}}o.\texttt{<unlock>}()^{\{k+1\}}\ \ \omega|\!\rangle\phi$$

$$\text{unlock}\ \frac{\neg path(i,p,tid) \Longrightarrow \langle\!|\alpha\ \ {}^{\{n-1\}}o.\texttt{<unlock>}()^{\{k+1\}}\ \ \omega|\!\rangle\phi}{\Longrightarrow \langle\!|\alpha\ \ \underbrace{{}^{\{n\}}o.\texttt{<unlock>}()^{\{k\}}}_{\text{at position } i \text{ in } p}\ \ \omega|\!\rangle\phi}$$

Fig. 5. The rule for lock release

A similar rule exists for the `<unlock>()` method (⇒ Fig. 5), which decreases the lock count and clears the locked-by status when the count reaches zero. For simplicity we do not clear the `<lockedby>` flag in the calculus, since it does not prevent further acquisition of the lock once `<lockcount>` has reached zero.

Programmers use locking protocols (besides thread-local data) to enforce atomicity of code sections. The easiest way to prove lock-based atomicity with our calculus is by using the invariant rule. An example of this is given in Section 8.3.

Recognising deadlock The presence of locking opens a possibility for deadlock. This can be modelled in the logic either as "normal" termination (the successor state is the current state) or as non-termination (there is no successor state). We have decided to model deadlock as normal termination. Nonetheless, it is still very easy to discern a deadlocked state from a run to completion by checking the final program configuration in the proof. In case of deadlock, the desired postcondition would also still have to hold, even if the program becomes disabled prematurely.

6.6. An Invariant Rule

So far, we have used induction for verifying full programs. In the following we present a complementary rule invariant, 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 given in Figure 6, but we need a notational device first.

Definition 6.1 (Step update). In the following rules, we use $\{\Delta pos\}$ as a notational abbreviation for the *step update* $\{pos(i) := pos(i) - 1 \,||\, pos(i+1) := pos(i+1) + 1\}$. The step update reflects the movement of one thread in the thread configuration. The abbreviation was not required in the rules given so far, as the relevant part of the configuration could be inlined at an explicitly given program position.

As before, $size(p)$ is the number of positions in the program p, $p_i^{*(tid(i))}$ is the sequential instantiation (⇒ Def. 3.4) of the atomic program at position i, and the id of the thread executing the instantiation $tid(i) = \pi_i(Post(i)+1)$.

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 no thread is longer enabled. What follows are $size(p)$

$$\Gamma \Longrightarrow \mathscr{U}\, INV,\ \Delta$$

$$INV,\ \mathscr{P} = \bot \Longrightarrow \phi$$

$$\vdots$$

$$INV,\ path(i,p,tid(i)),\ enabled(i) \Longrightarrow \langle\!\![\underline{p_i^{*(tid(i))}}]\!\!\rangle \{\Delta pos\} INV$$

$$\vdots$$

$$\mathsf{invariant}\ \frac{}{\Gamma \Longrightarrow \mathscr{U}\, \langle\!\![p]\!\!\rangle \phi,\ \Delta}\ (*)$$

i ranges over $1 \ldots size(p)$

Fig. 6. The concurrent invariant rule

premisses—one for each position in the program—stating that the "sequential" execution of the atomic statement at this position preserves the invariant. For each position we can assume its enabledness predicate and the corresponding path condition.

In its structure, the invariant rule resembles the standard sequential loop invariant rule, but with a few differences. First, while a loop only has one degree of freedom (the execution of the loop body), a concurrent program has one degree of freedom for each potentially enabled position. Every executed statement brings the system into a new state, and, thus, has to be shown as invariant-preserving. Second, the concurrent invariant formula can—and most probably will—contain control variables, which correspond to the loop counter. Third, our invariant rule is sound for the diamond modality even without a special termination argument. The only potential sources of non-termination are loops, which we assume as atomic, and the sequential calculus fragment is sound and complete for these. For this reason, the above invariant rule is also not needed for the completeness of the concurrent calculus.

6.7. Remarks on Calculus Soundness

The soundness of a verification calculus—together with the adequacy of the underlying programming language theory—is an issue of great importance. We have validated our calculus (and its implementation) by testing it as explained below. As with the sequential calculus of KeY, we have not performed a formal or even mechanised soundness proof. The reason for this decision is an effort trade-off, which we explain in detail in [Kle09].

We did, however, state a formal semantics of the logic. Among other things, the semantics defines the scheduler axioms, which are used by the calculus. In fact, we stated two versions of semantics: one with explicit thread ids (⇒ Sect. 3) and one with permutations (⇒ Sect. 5). This approach has helped us to separate concerns present in developing a general program logic with a deterministic scheduler and, later, a logic with symmetry reduction. It is an interesting question why the latter logic correctly simulates the former.

The key to answering this question is the configuration concretisation function (⇒ Def. 5.3) and the scheduler decomposition equality (4). The configuration concretisation function explains how every configuration with permutations can be translated into a configuration with concrete tids. The scheduler decomposition explains the same for the scheduler function. Both translations are quite simple, and allow us to fall back on many common definitions in both logics.

Ultimately, of course, there can be no formal proof that any of these semantics is adequate w.r.t. the Java Language Specification or the implementation of any given compiler and JVM. In this light, testing is necessary for obtaining a reliable reasoning system. The sequential KeY calculus is automatically tested with the compiler test suite Jacks [Jac] on a regular basis. The suite is a collection of intricate programs covering many difficult features of the Java language. These programs are executed with the symbolic execution engine of KeY and the output is compared to the reference provided by the suite.

For reasons probably related to nondeterminism, such test suites do not include multi-threaded programs. We are testing MODL on our own collection of very simple programs capturing typical multi-threading situations (threads competing for the same lock, for different locks, in a deadlock, etc.). We are using programs exhibiting both desired and undesired outcomes from a programmer's point of view. The programs are symbolically executed in KeY and the

$$\frac{\begin{array}{rl} \forall k.\ (1 \le k \le \mathit{size}(p) \rightarrow \neg \mathit{enabled}(k)) \Longrightarrow \phi & \\ \vdots & \\ \mathit{path}(i,p,\mathit{tid}(i)),\ \mathit{enabled}(i) \Longrightarrow & \langle\!\underline{[p_i^{*(\mathit{tid}(i))}]}\!\rangle\{\Delta pos\}\langle\!\langle p\rangle\!\rangle\phi \\ \neg\mathit{path}(i,p,\mathit{tid}(i)),\ \mathit{enabled}(i) \Longrightarrow & \{\Delta pos\}\langle\!\langle p\rangle\!\rangle\phi \\ \vdots & \end{array}}{\Longrightarrow \langle\!\langle p\rangle\!\rangle\phi}\ \textsf{step (impl.)}$$

i ranges over $1 \ldots \mathit{size}(p)$

Fig. 7. Implementation of the step rule

proof attempts are manually inspected. Since the calculus is based on symbolic execution, the results are significant even with minimal specifications.

It would be interesting to investigate how to automate such tests. An automated test suite of concurrency microbenchmarks would be a very useful tool for ensuring reliability of verification systems and a great benefit to the field.[10]

7. Implementation

The changes w.r.t. a stock KeY system amount to about 3200 lines of code in 56 files. The greatest technical difficulty by far was a generalisation of the rule application engine. From the very beginning the KeY system was designed to apply program-manipulating rules only at the beginning of a program. This limitation had to be lifted in order to support multi-threaded execution.

Specification Verification problems are specified in Dynamic Logic and input to the prover as so-called dot-key files [BHS07]. The new keyword `\local` distinguishes thread-local from static variables in declarations. Thread configurations are specified with updates to the non-rigid function `pos`.

The calculus rules For efficiency reasons, the step rule is implemented slightly differently from the formulation shown in Section 6.3. There is no premiss $\Longrightarrow \mathscr{P} = k$. Instead the implementation follows the pattern of the invariant rule and automatically performs a case distinction over all positions. The rule is shown in Figure 7. Per position at least two subgoals are generated: one for the positive and one for the negative path condition. In the positive case, the sequential instantiation of the statement at the position is tackled. The effect may include generating an update, locking an object, or producing further case distinctions, e.g., to check for a null reference. In the negative case, only the thread configuration changes. An additional subgoal is added for the case that all positions are disabled.

Automation Proof search is automated by the usual strategies of the KeY prover. We have extended the main strategy with a further parameter controlling how many times the step rule is to be applied automatically: (a) never, (b) unlimited, or (c) until some thread becomes disabled. The third setting is especially useful when performing induction proofs. In all cases, step is executed with very low priority, i.e., only after no other rules are applicable and the state description has been simplified as far as possible.

We have also implemented an unfolding strategy that pools all rules for program unfolding (⇒ Sect. 6.2). This strategy is only used for preparing proof obligations and is not active during proof search.

[10] The existing catalogues and samplers of multi-threading bug patterns such as [BJ09, EU04] have a related but slightly different focus. They are targeted primarily towards testing the efficiency of bug-finding tools.

```
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;
}
```

Fig. 8. Excerpt from the `StringBuffer` class

```
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>();
```

Fig. 9. Method `append(char c)` unfolded

8. Verifying Full Functional Correctness of Appending to a StringBuffer

We have applied our system to verify the full functional correctness of a method of the `StringBuffer` class in the 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 `StringBuffer` source code shipped by SUN with the JDK 1.4.2 (shown in Figure 8). 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.

We now describe the verification process as carried out with KeY.

8.1. Specification

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

$$\langle \underline{\texttt{strb = new StringBuffer();}} \rangle \, \forall n. \left(n > 0 \rightarrow \langle {}^{\{n\}}\texttt{strb.append(c);}^{\{0\}} \rangle \texttt{strb.count} = n \wedge \forall k. \, \left(0 \leq k < n \rightarrow \texttt{strb.value}[k] = \texttt{c}(\pi_1(k+1)) \right) \right) ,$$

where `strb` is a static variable of type `StringBuffer`[11] and `c` is a thread-local `char` variable.

[11] The MODL program definition calls for a local variable here, but the calculus is more liberal in this regard. We used this liberty to write a simpler proof obligation while still achieving the same effect.

Plainly speaking: if n threads are concurrently performing an append on a freshly created shared `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.

After symbolic execution of the `StringBuffer` creation (in the sequential diamond) and Skolemisation, KeY has reduced the original conjecture to:

$$\mathit{Init} \wedge n_0 > 0 \rightarrow \langle \texttt{\{}_{n_0}\texttt{\}strb.append(c);\{}_{0}\texttt{\}} \rangle \texttt{strb.count} = n_0 \wedge \\ \forall k.\ \big(0 \leq k < n_0 \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))\big)\ , \quad (11)$$

where n_0 is a fresh integer constant and *Init* is a formula capturing the state after `StringBuffer` creation. *Init* is shorthand for:

$$\texttt{strb} \neq \texttt{null} \wedge \texttt{strb.<lockcount>} = 0 \wedge \texttt{strb.count} = 0 \wedge \\ \texttt{strb.value} \neq \texttt{null} \wedge \texttt{strb.value.length} = 16 \wedge \\ \texttt{strb.value} \neq \texttt{jchar[]::<get>(jchar[].<nextToCreate>)}\ .$$

The cryptic last conjunct states that the current `value` array is not aliased to the next `char` array to be created and is owed to the way KeY deals with instance creation.

8.2. Unfolding

To proceed with verification, we apply the unfolding strategy (⇒ Sect. 6.2) to the `append()` call. The implementation is inlined (together with the `expandCapacity()` method), and fresh local variables are introduced to eliminate side effects and make explicit the atomicity granularity of the code. The result is shown in Figure 9, 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 `arraycopy(`src`,`$srcPos$`,`$dest$`,`$destPos$`,`len`)` into a quantified update

$$\{\texttt{for}\ l; 0 \leq l < \mathit{len}; \mathit{dest}\,[\mathit{srcDest} + l] := \mathit{src}\,[\mathit{srcPos} + l]\}\ ,$$

which is a concise way to express a number of updates at once.

8.3. Establishing Atomicity

A method or code block is *atomic* if its execution is not affected by and does not interfere with concurrently-executing threads. More formally, a code block β is atomic if for every program execution with final state s, there is some equivalent (i.e., also ending in s) execution, where β is executed without interruption [FF04].

There are two main reasons for us wanting to establish atomicity of code sections:

- One reason roots in the limitation of the MODL calculus that all loops must be atomic (i.e., appear within atomic blocks). In real Java programs, atomicity of code sections is implemented implicitly with locking or thread-local data encapsulation. Thus, it is necessary to prove that every such implementation is indeed correct, and no unsoundness is introduced by putting loops into explicit atomic blocks.
- The second reason is to coarsen the interference granularity of programs and simplify reasoning about their concurrent behaviour. It is often useful to separate concerns, i.e., to establish atomicity of code sections first, and then use this fact in further proof of functional correctness.

In MODL, we prove a sufficient condition for atomicity. A code block β is atomic, if the following holds throughout program execution:

$$0 \leq \sum_{i \in C(\beta)} \mathit{pos}(i) \leq 1\ , \quad (12)$$

where $C(\beta)$ is the set containing at least (a) all program positions of β and (b) all positions that access the same shared

state as β[12]. This condition ensures that whenever some thread could execute a statement potentially interfering with β, β has either not yet started or has already finished.

We now use the invariant rule to show that `append()` can only be executed by one thread at a time (on the same `StringBuffer` object) and thus establish its atomicity. We want to show that the configuration never has more than one thread between its second and the last but one position:

$$0 \leq N \leq 1, \text{ where } N = \sum_{i=2}^{q} pos(i) \ .$$

For the proof to succeed, the above has to be strengthened to[13]

$$INV \equiv 0 \leq N \leq 1 \wedge \big(N > 0 \leftrightarrow \texttt{strb.<lockcount>} > 0\big).$$

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. To show this, we prove

$$INV,\ path(1,p,tid(1)),\ enabled(1) \Longrightarrow \langle\!\langle \underline{\texttt{strb.<lock>();}^{*(tid(1))}} \rangle\!\rangle \{pos(1) := pos(1)-1 \,||\, pos(2) := pos(2)+1\} INV \ .$$

We perform a case split:

- If the lock is available (i.e., `strb.<lockcount>` $= 0$), then INV in the antecedent simplifies to $N = 0$. After the symbolic execution of lock acquisition and update application, both N and `strb.<lockcount>` are equal to 1, and thus INV holds again.
- If the lock is not available, then we get a contradiction in the antecedent between `strb.<lockcount>` > 0 and $enabled(1)$. The goal is discharged.

Per this invariant, once a thread has entered the method it will run to completion without interference. Thus, the method is atomic, and we can elide locking, replacing it by an atomic block. Our conjecture becomes:

$$Init \wedge n_0 > 0 \rightarrow \langle^{\{n_0\}} \ll \texttt{strb.append1(c);} \gg^{\{0\}} \rangle \texttt{strb.count} = n_0 \wedge \forall k.\ \big(0 \leq k < n_0 \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))\big) \ , \quad (13)$$

where the method `append1(c)` (shown here folded) is identical to `append(c)` save for the removed locking operations.

8.4. 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 $n_0 < 2^{31}$. 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 the presence of proper synchronization.

Since there is no way to fix the method, we have to amend the conjecture with a pre-condition limiting the value of n_0. Please note that this is not due to a limitation of our proof method. We now prove full functional correctness

[12] This information is typically available in form of modifies/reads clauses for methods.
[13] For clarity, we elide the irrelevant details related to potentially reentrant locking.

with the following, quite natural invariant:[14]

$$\begin{aligned} INV \equiv\ & pos(1)+pos(2) = n_0 \wedge n_0 < 2^{31} \wedge pos(2) \geq 0 \wedge \\ & \texttt{strb.count} = pos(2) \wedge \\ & \forall k.\ (0 \leq k \wedge k < pos(2) \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))) \wedge \\ & \texttt{strb} \neq \texttt{null} \wedge \texttt{strb.value} \neq \texttt{null} \wedge \\ & \texttt{strb.value.length} \geq \texttt{strb.count} \wedge \\ & \texttt{strb.value} \neq \texttt{jchar[]::<get>(jchar[].<nextToCreate>)} \ . \end{aligned}$$

Applying the invariant rule to (13) produces three premisses.

Premiss 1: invariant initially valid In this premiss we need to prove the sequent $\Gamma \Longrightarrow \mathscr{U} INV, \Delta$. Here, Γ contains just $Init \wedge (n_0 > 0) \wedge (n_0 < 2^{31})$, and Δ is empty. The update $\mathscr{U}$ is given by the thread configuration of the original program. The formula $\langle^{\{n_0\}} \ll \texttt{strb.append1(c);} \gg^{\{0\}} \rangle \phi$ is shorthand for

$$\{pos(1) := n_0 \,||\, pos(2) := 0\}\ \langle \ll \texttt{strb.append1(c);} \gg \rangle \phi \ .$$

The proof obligation is thus:

$$\begin{aligned} & Init \wedge n_0 > 0 \wedge n_0 < 2^{31} \Longrightarrow \\ & \quad \{pos(1) := n_0 \,||\, pos(2) := 0\}\ \big(pos(1)+pos(2) = n_0 \wedge n_0 < 2^{31} \wedge pos(2) \geq 0 \wedge \\ & \qquad\qquad \texttt{strb.count} = pos(2) \wedge \\ & \qquad\qquad \forall k.\ (0 \leq k \wedge k < pos(2) \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))) \wedge \\ & \qquad\qquad \texttt{strb} \neq \texttt{null} \wedge \texttt{strb.value} \neq \texttt{null} \wedge \\ & \qquad\qquad \texttt{strb.value.length} \geq \texttt{strb.count} \wedge \\ & \qquad\qquad \texttt{strb.value} \neq \texttt{jchar[]::<get>(jchar[].<nextToCreate>)} \big) \end{aligned}$$

The quantifier in the succedent has an empty range (due to the update $pos(2) := 0$), and further basic rewriting renders the sequent proved. The KeY system finds the proof automatically in 67 steps.

Premiss 2: invariant implies postcondition upon termination of all threads In this premiss we need to prove the sequent $INV,\ \mathscr{P} = \bot \Longrightarrow \phi$, where ϕ is the postcondition. Since the atomic block is the only position, $\mathscr{P} = \bot$ is equivalent to $pos(1) = 0$ (per Axiom (7)). The proof obligation is thus:

$$\begin{aligned} & pos(1)+pos(2) = n_0 \wedge n_0 < 2^{31} \wedge pos(2) \geq 0 \wedge \\ & \texttt{strb.count} = pos(2) \wedge \\ & \forall k.\ (0 \leq k \wedge k < pos(2) \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))) \wedge \ldots, \\ & pos(1) = 0 \Longrightarrow \\ & \quad \texttt{strb.count} = n_0 \wedge \\ & \quad \forall k.\ \big(0 \leq k < n_0 \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(\pi_1(k+1))\big) \ , \end{aligned}$$

This sequent is easily discharged, since $pos(1)+pos(2) = n_0$ together with $pos(1) = 0$ implies $pos(2) = n_0$. The KeY system finds the proof automatically in 108 steps.

Premiss 3: invariant preservation In this premiss we need to prove

$$INV,\ path(1,p,tid(1)),\ enabled(1) \Longrightarrow \langle\!\underline{[p_1^{*(tid(1))}]}\!\rangle \{pos(1) := pos(1)-1 \,||\, pos(2) := pos(2)+1\} INV$$

which is a purely sequential proof obligation. After expanding the definitions, the path condition simplifies to *true* and the predicate $enabled(1)$ to $pos(1) > 0$. We also expand the definition of sequential program instantiation, obtaining the goal

$$\begin{aligned} & INV,\ pos(1) > 0 \Longrightarrow \\ & \qquad \langle\!\underline{[\ll \texttt{strb.append1(c}(\pi_1(pos(2)+1))\texttt{);} \gg]}\!\rangle \{pos(1) := pos(1)-1 \,||\, pos(2) := pos(2)+1\} INV \end{aligned}$$

This goal is the most difficult to prove, since it requires symbolic execution of the method, reasoning about Java-faithful arithmetic, and quantifier instantiation. The KeY system finds the 2898-step long proof automatically in about 30 seconds.

[14] It is also possible to use induction in a manner similar to Example 6.1.

```
1  public synchronized StringBuffer append(StringBuffer sb) {
2      if (sb == null) {
3          sb = NULL;
4      }
5
6      int len = sb.length();              // (A)
7      int newcount = count + len;
8      if (newcount > value.length)
9          expandCapacity(newcount);
10     sb.getChars(0, len, value, count); // (B)
11     count = newcount;
12     return this;
13 }
```

Fig. 10. Atomicity failure in `StringBuffer`

8.5. Towards Thread-Modularity and Further Issues with java.lang.StringBuffer

In this section we show that MODL can be combined with rely-guarantee-style [Jon81] reasoning to achieve thread-modular verification. We demonstrate the approach using another example from the `StringBuffer` class.

The example The `StringBuffer` class contains another method, `append(StringBuffer sb)`, for appending the content of the `StringBuffer` `sb` to the current `StringBuffer`. The code of this method is shown in Figure 10. In a closed-world setting, a specification similar to (11) can be verified for this method as well.

The method has two critical points: when the length of `sb` is queried at (A) and when the characters are actually copied at (B). The problem with the code is that nothing prevents some other thread operating on `sb` to be scheduled between the execution of (A) and (B). The intruding thread may end up removing characters from `sb`; the length read at (A) becomes stale and an attempt to copy no-longer-existing characters at (B) produces an exception.

Note that this scenario technically does not constitute a data race. The methods `length()`, `getChars()`, and `delete()` (potentially invoked by another thread) are synchronized, and thus, all access to shared data of `sb` is protected by locks. The problem is rather that the lock is released and then re-acquired within `append(sb)`, violating the application-specific atomicity policy. One can speculate that this was done for performance reasons.

A step rule for thread-modular verification We let an MODL program be accompanied by *an environment*. The environment is specified by its signature $\bar{\mathtt{e}}$, which is a vector of program variables that the environment may modify, as well as a rely condition Φ, which is a formula restricting the actions that the environment may perform. We require that the rely condition is transitive. This way a single environment "step" can model an arbitrary number of subsequent environment actions.

We extend the step (impl.) rule from Figure 7 as shown in Figure 11. For each program position, we have added an extra premiss capturing potential interference from the environment. We have also extended the first premiss, demanding that ϕ is preserved by the environment once all threads have run to completion.

The example: specification For our example, we start with an environment that allows almost arbitrary modifications to the StringBuffer objects referenced by the thread-local parameters `sb`. The rely condition

$$\forall i.\,(\mathtt{sb}(i)\mathtt{.<lockcount>} = 0 \wedge sb'(i) \neq \mathtt{null} \wedge sb'(i)\mathtt{.<lockcount>} = 0) \tag{14}$$

specifies that StringBuffer objects may only be mutated when their lock is free, the new value of the `sb` reference must be non-null and the object lock is available after the mutation.

The example: atomicity In the presence of an environment, the append method can no longer be modelled as a single atomic block. Instead, we can split the method body into two atomic blocks encompassing the statements in the lines 2–6, and 7–12 respectively. The soundness of such coarsening can be proven in a fashion analogous to (12).

For the atomicity proof, the environment can be modelled as an always-enabled "abstract" thread, whose updates we split into two parts. One part manipulates the StringBuffer of the "current" thread while respecting the locking discipline (as specified in the rely condition). The other part manipulates only the objects not reachable from the current thread. To ensure this separation, we strengthen the precondition of the program to demand that to-be-appended StringBuffer objects are not aliased. We note that the obtained atomic blocks are larger than the scope of full synchronization (line 6 and line 10 resp.), as the other statements only operate on thread-local data.

$$\text{step-env}\ \frac{\begin{array}{rl}\forall k.\ (1 \le k \le size(p) \rightarrow \neg enabled(k)),\ \Phi(\bar{\mathtt{e}}, \vec{e}\,') \Longrightarrow & \{\bar{\mathtt{e}} := \vec{e}\,'\}\phi \\ & \vdots \\ path(i,p,tid(i)),\ enabled(i) \Longrightarrow & \langle\!\underline{[\![p_i^{*(tid(i))}]\!]}\rangle\{\Delta pos\}\langle\!\underline{[\![p]\!]}\rangle\phi \\ \Phi(\bar{\mathtt{e}}, \vec{e}\,'),\ path(i,p,tid(i)),\ enabled(i) \Longrightarrow & \{\bar{\mathtt{e}} := \vec{e}\,'\}\langle\!\underline{[\![p_i^{*(tid(i))}]\!]}\rangle\{\Delta pos\}\langle\!\underline{[\![p]\!]}\rangle\phi \\ \neg path(i,p,tid(i)),\ enabled(i) \Longrightarrow & \{\Delta pos\}\langle\!\underline{[\![p]\!]}\rangle\phi \\ & \vdots \end{array}}{\Longrightarrow \langle\!\underline{[\![p]\!]}\rangle\phi}$$

$\vec{e}\,'$ is a fresh symbol

i ranges over $1 \ldots size(p)$

Fig. 11. The environment-aware step-env rule

The fact that the method must be split into two atomic blocks already represents a red flag for its correctness.

The example: symbolic execution After atomicity coarsening, we can perform symbolic execution in the presence of environment. This promptly uncovers the problem of the stale length causing an exception. To avoid the crash, it is necessary either to fix `append()` by extending the scope of locking `sb`, or to limit the actions of the environment, e.g., forbidding removal of characters from StringBuffer objects. Then, it is possible to verify the program using the invariant rule or induction just as in Section 8.4. The exact property that can be verified depends, of course, on the strength of the environment specification: the resulting StringBuffer may contain character sequences originating from the environment.

9. Conclusion and Future Work

We have shown that it is possible to execute multi-threaded programs symbolically while taking full data into account. This was made possible by an extension of the technique of symmetry reduction. By employing an explicit scheduler function, our calculus can track full information about state quite efficiently, but permits abstraction for further improvement. Nondeterministic scheduler choice is adequately modelled by a deterministic function that has a fixed but unknown value. This formalisation enables efficient deduction. Relating different runs of the scheduler is possible by incorporating different "don't-knows."

We have performed a mechanised proof of full functional correctness of a piece of real production code. The proof demonstrates how multi-threading can result in subtle bugs even in correctly synchronized programs. Currently, we are working on extending the covered Java fragment, in particular on lifting the loop atomicity limitation. Incorporating the iteration counter as a parameter of the thread choice functions seems promising in this regard.

It is known that the efficiency of a verification system is bounded to a great degree by the compositionality of reasoning it offers. We have shown how to combine MODL with rely-guarantee-style reasoning to achieve thread-modular verification. We also expect no problems in incorporating standard techniques (e.g., [GS02, RDF+05]) for method-modular verification of multi-threaded Java programs.

References

[AdBdRS05] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. An assertion-based proof system for multithreaded Java. *Theor. Comp. Sci.*, 331(2–3):251–290, 2005.

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[Ash75] Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, 1975.

[BDRS02] Michael Balser, Christoph Duelli, Wolfgang Reif, and Gerhard Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BJ09] Jeremy S. Bradbury and Kevin Jalbert. Defining a catalog of programming anti-patterns for concurrent Java. In *Proc. of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09)*, pages 6–11, 2009.

[BP06] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006.

[BS05] Bernhard Beckert and Steffen Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4):423–442, 2005.

[dB07] Frank S. de Boer. A sound and complete shared-variable concurrency model for multi-threaded Java programs. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4468 of *LNCS*, pages 252–268. Springer, 2007.

[DRB02] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Towards the automated verification of multithreaded Java programs. In Joost-Pieter Katoen and Perdita 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.

[EU04] Yaniv Eytani and Shmuel Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proceedings, 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. IEEE Computer Society, 2004.

[FF04] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267. ACM, 2004.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.

[GS02] Aaron Greenhouse and William 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.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[Jac] Jacks is an automated compiler killing suite. At `http://www.sourceware.org/mauve/jacks.html`.

[Jon81] Cliff B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.

[JP01] Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. In *Proceedings, 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 284–299. Springer, 2001.

[JSPS06] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Zhiming Liu and Jifeng He, editors, *8th International Conference on Formal Engineering Methods, ICFEM, Macao, China, Proceedings*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Kle04] Vladimir 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.

[Kle09] Vladimir Klebanov. *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Department of Computer Science, Universität Koblenz-Landau, 2009.

[KPV03] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Proceedings, 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.

[MP91] Zohar Manna and Amir 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.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certificationof Java/Java Card programs annotated in JML. *J. Log. Algebr. Program.*, 58(1–2):89–106, 2004.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Pel87a] David Peleg. Communication in concurrent dynamic logic. *J. Comput. Syst. Sci.*, 35(1):23–58, 1987.

[Pel87b] David Peleg. Concurrent dynamic logic. *J. ACM*, 34(2):450–479, 1987.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In D. Swierstra, editor, *Proceedings, ESOP '99*, LNCS 1576. Springer, 1999.

[RDF+05] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP*, LNCS 3586, pages 551–576. Springer, 2005.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM.

[RDHI03] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software. In *Proceedings SoftMC 2003, Workshop on Software Model Checking, ENTCS 89*, 2003.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[VP07] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of Rely/Guarantee and Separation Logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Proceedings 18th International Conference on Concurrency Theory (CONCUR 2007), Lisbon, Portugal*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[Yah01] Eran 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.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008*, pages 349–361. ACM, 2008.

Selected Symbols

Symbol	Meaning	See
$(*)$	indicates rules from which the sequent context cannot be omitted	Sect. 6.1
$\bot$	no thread or no position enabled	
$[\cdot]$	box modality	Def. 3.10, 4.1
$\langle\cdot\rangle$	diamond modality	Def. 3.10, 4.1
$\langle[\cdot]\rangle$	schematic modality standing for either a diamond or a box	Sect. 6.1
$\{\Delta pos\}$	step update abbreviation	Def. 6.1
$enabled(i)$	position enabledness	Def. 5.4
$enabled(s)$	statement enabledness	Def. 3.8
$enabled(s,t)$	statement enabledness for a thread	Def. 3.9
$\mathscr{K}$	Kripke structure	Sect. 4.1
$\mathscr{P}$	position choice function (scheduler component)	Sect. 5.4
$path(k,p,tid)$	path condition of position k in program p for thread tid	Def. 4.4
π_i	thread choice function at position i (scheduler component)	Def. 5.1
$pos(i)$	(number of) threads currently available for scheduling at position i	Def. 3.5, 5.1
pos_γ	configuration concretisation	Def. 5.3
$Post(i)$	number of threads past position i	Def. 5.2
ρ	transition relation for concurrent programs	Sect. 4
ρ_1	transition relation for sequential programs	Sect. 4
$\mathscr{S}$	set of states (of a Kripke structure)	Sect. 4.1
$s[\![u]\!]$	state achieved from state s via update u	Def. 4.2
$sched$	scheduler function	Def. 3.7
$size(p)$	total number of positions in a program p	Sect. 3.5
$\mathscr{T}$	set of thread identifiers	Sect. 3.1
$\mathscr{U}$	sequence of updates	Sect. 6.1