Model Checking Multithreaded Programs with Asynchronous Atomic Methods

Koushik Sen and Mahesh Viswanathan

Department of Computer Science,
University of Illinois at Urbana-Champaign
{ksen, vmahesh}@cs.uiuc.edu

Abstract. In order to make multithreaded programming manageable, programmers often follow a design principle where they break the problem into tasks which are then solved asynchronously and concurrently on different threads. This paper investigates the problem of model checking programs that follow this idiom. We present a programming language SPL that encapsulates this design pattern. SPL extends simplified form of sequential Java to which we add the capability of making asynchronous method invocations in addition to the standard synchronous method calls and the ability to execute asynchronous methods in threads atomically and concurrently. Our main result shows that the control state reachability problem for finite SPL programs is decidable. Therefore, such multithreaded programs can be model checked using the counterexample guided abstraction-refinement framework.

1 Introduction

Multithreaded programming is often used in software as it leads to reduced latency, improved response times of interactive applications, and more optimal use of processing power. Multithreaded programming also allows an application to progress even if one thread is blocked for an I/O operation. However, writing correct programs that use multiple threads is notoriously difficult, especially in the presence of a shared mutable memory. Since threads can interleave, there can be unintended interference through concurrent access of shared data and result in software errors due to data race and atomicity violations.

Therefore, programmers writing multithreaded code, often adhere to a design idiom where the computational problem is broken up into *tasks* which are then assumed to be finished asynchronously, concurrently, and atomically. Specifically, threads during their execution may send tasks or events or asynchronous messages to other threads. If a thread is busy completing a task, the messages sent to it get added to a pool of tasks associated with the thread. When the thread has completed the current task, it takes out a task from its pending pool and starts processing it concurrently with other threads. If the pool is empty the thread waits for a new task or event or message. Even though these asynchronous tasks are executed concurrently on different threads, an underlying assumption is that these tasks will be executed atomically. This is often ensured through various synchronization primitives, such as locks, mutexes, semaphores, etc.

For example, in the Swing/AWT subsystem of Java, a non-GUI thread is not allowed to make any direct changes to the user interface represented by a Swing

© Springer-Verlag Berlin Heidelberg 2006

object. Instead such a thread submits the request to the `EventQueue` by calling `SwingUtilities.invokeLater(runnable)`. The thread associated with the Swing event queue handles these requests one by one atomically. This ensures that the user interface operations are performed in a non-interfering way and the user interface has a consistent state and look. Another context where this design paradigm is widely prevalent is multithreaded web servers. When a page request is sent to a web server, the web server posts the request to a request queue. If there is a free thread in the finite thread pool of the web server, the free thread removes a request from the request queue and starts processing the request. The use of threads ensures that multiple requests to the web-server can be served concurrently. Moreover, synchronization primitives are used to ensure that the threads do not interfere with each other. Another application area where this paradigm is used is embedded software which is naturally event-driven. Finally, multithreaded transaction servers for databases also view transactions as asynchronous requests that are served by the different threads of the server concurrently. Since requests in this context are transactions, the server ensures that the service of a transaction satisfies the ACID (atomicity, consistency, isolation, durability) property.

The prevalence of this design idiom has also been observed by Allen Holub [17] in his book "Taming Java Threads". In this book, Holub points out that programmers classify method invocations or messages into two categories: synchronous messages and asynchronous messages. The handler for synchronous messages doesn't return until the processing of the message is complete. On the other hand, asynchronous messages are processed possibly by a different thread in the background some time after the message is received. However, the handler for asynchronous messages returns immediately, long before the message is processed.

In this paper, we investigate the verification of programs written adhering to this design principle. We introduce a simple programming language, called SPL, that encapsulates this design goal. It is a simplified form of sequential Java to which we add the capability of making asynchronous method invocations in addition to the standard synchronous method calls and the ability to dynamically create threads. We define its semantics in terms of concurrently executing threads. We then observe that the requirement that asynchronous methods execute atomically, allows us to reason about the program using a new semantics wherein the threads service these asynchronous method invocations serially.

The analysis of SPL programs with respect to the serialized semantics can then proceed by following the popular methodology of software model checking [29,2,15], where the program is first automatically abstracted using boolean predicates into one that has finitely many global states, and the abstracted program is then model checked. The results of the model checking are then used to either demonstrate a bug/correctness of the program, or used to refine the abstraction.

The success of the software model checking framework depends upon the model checking problem for SPL programs with finitely many global states being decidable. We first observe that the serial semantics ensures that the local stack of at most one thread is non-empty at any time during the execution; the semantics of such programs can thus be defined using only one stack. We introduce *multi-set pushdown systems* (MPDS) to model such finite SPL programs. MPDSs have finitely many control states, one unbounded stack to execute recursive, synchronous methods, and one unbounded bag to store the asynchronous method invocations. The main restriction that is imposed on such systems is that messages from the bag be serviced only when the stack is empty, a consequence of our atomicity requirements. Our main result is that the problem of control state reachability of MPDSs is decidable, thus demonstrating that SPL programs can be analyzed in the counterexample guided abstraction-refinement framework.

The rest of the paper is organized as follows. Next, we discuss closely related work and place our results in context. Section 2 introduces notation, definitions and classical results used in proving our results. The simple parallel language (SPL) for multithreaded programming is presented along with its semantics in Section 3. We investigate the verification problem in Section 4 and conclude by giving a lower bound. Due to lack of space, we defer some of the proofs to [31].

Related Work. Model-checking algorithms and tools [29,2,15] for single-threaded programs with procedures based on predicate abstraction have been developed. These model checkers use the fact that the reachable configurations of pushdown systems are regular [1,13]. Ramalingam [28] showed that verification of concurrent boolean programs is undecidable. As a consequence, approximate analysis techniques that over-approximate [4] and under-approximate [26,3] the reachable states have been considered, as have semi-decision procedures [25]. Note that the algorithm in [25] can be shown to terminate if the whole execution of a thread is assumed to be a transaction. Other techniques [7,16,8] try verifying each thread compositionally, by automatically abstracting the environment. Finally, the KISS checker [27] for concurrent programs simulates the executions of a concurrent program by the executions of a sequential program, where the various threads of the concurrent program are scheduled by the single stack of the sequential program. It is worth mentioning [27] first proposed the use of a single stack to model executions of multithreaded software. Though complete, the KISS checker is not sound.

There has also been considerable effort in characterizing concurrent systems with finitely many global states for which the reachability analysis is unknown to be decidable. Starting from the work of Caucal [6] and Moller [22], where purely sequential and purely parallel processes were considered, hierarchies of systems have been defined. Mayr [21] gave many decidability and undecidability results based on a unified framework. Among the models that allow both recursion and dynamic thread creation, most disallow any form of synchronization between the threads [11,30,20,23]. More recently, the model of *constrained dynamic pushdown networks* (CDPN) [5] was introduced which allowed for thread creation and limited forms of synchronization. CDPNs have a more sophisticated means to synchronize, but they limit synchronization only between a parent thread and its descendants. Our model of MPDS, allows dynamic thread creation and limits context switches to happen only when asynchronous methods have finished execution. Thus, MPDSs and CDPNs are incomparable and apply to different multi-threaded programs.

2 Preliminaries

Multi-sets and Strings. Given a finite set Σ, the collection of all finite multi-sets with elements in Σ will be denoted by $M_\omega[\Sigma]$. We say $a \in M$ if a is an element of multi-set

M. For multi-sets M and M', $M \cup M'$ is the multi-set union of M and M', and $M \setminus M'$ the multi-set difference between M and M'. We use $\emptyset$ to denote the empty multi-set. Recall that Σ^* is the collection of all finite strings over the alphabet Σ, with ϵ being the empty string. Given two finite strings w and w', we will denote their concatenation by ww'. For a string w, $\mathbb{M}(w)$ will denote the multi-set formed from the symbols of w. For example, if $w = aaba$, then $\mathbb{M}(w) = \{a, a, a, b\}$. Finally, for $L \subseteq \Sigma^*$, $\mathbb{M}(L) = \{\mathbb{M}(w) \mid w \in L\}$.

Well-quasi-orderings. Recall that a *quasi-ordering* $\leq$ over a set X, is a binary relation that is reflexive and transitive. Given a quasi-ordering $\leq$, an *upward closed set* $U \subseteq X$ is a set such that if $x \in U$ and $x \leq y$ then $y \in U$. For a set $S \subseteq X$, the smallest upward closed set containing S will be denoted by $\text{CL}(S)$, i.e., $\text{CL}(S) = \{x \mid \exists y \in S.\, y \leq x\}$. For a set S, the minimal elements in S is $\text{MIN}(S) = \{x \mid \forall y \in S.\, y \not\leq x\}$.

A quasi-ordering $\leq$ over X is said to be a *well-quasi-ordering* (wqo) if for any infinite sequence $x_1, x_2, x_3, \ldots$ of elements in X, there exist indices i, j such that $i < j$ and $x_i \leq x_j$. We now recall some well-known observations about well-quasi-orderings [18,12].

Proposition 1. *For a wqo $\leq$ and any set $S \subseteq X$, $\text{MIN}(S)$ is finite.*

Proposition 2. *For a wqo $\leq$, any infinite increasing sequence $U_0 \subseteq U_1 \subseteq U_2 \subseteq \cdots$ of upward closed sets eventually stabilizes, i.e., there is a $k \in \mathbb{N}$ such that for all $i \geq k$ $U_i = U_k$.*

Pushdown Systems. A *pushdown system* (PDS) is $\mathcal{P} = (Q, \Gamma, \delta, q_0, \gamma_0)$, where Q is a finite set of states, Γ is a finite set of stack alphabets, $q_0 \in Q$ is the initial state, $\gamma_0 \in \Gamma$ is the initial stack configuration, and $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is the transition relation. The execution of a PDS can be described in terms of a transition system over configurations, which are $(q, w) \in Q \times \Gamma^*$. We say $(q_1, w_1\gamma) \longrightarrow (q_2, w_1 w_2)$ if there is a transition $((q_1, \gamma), (q_2, w_2)) \in \delta$. We say a configuration (q, w) is reachable iff $(q_0, \gamma_0) \longrightarrow^* (q, w)$, where $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$, and that a control state q is reachable iff (q, w) is reachable for some $w \in \Gamma^*$. It is well-known that the problem of control state reachability is decidable (see [13,1]); this is the content of the next theorem.

Theorem 1. *Given a PDS $\mathcal{P}$, checking if a control state q is reachable is decidable in $O(n^3)$ time, where n is the size of the PDS $\mathcal{P}$.*

3 Programming Language

We describe a simple parallel language SPL, which captures the essential concepts of multithreaded programs with asynchronous atomic methods. The SPL language is a simplified form of the sequential Java language. Similar to Java, the SPL language supports objects. In addition to definition of classes, we allow the definition of a special type called **thread**. Instances of a **class** is called an object and instances of a **thread** is called a thread object. A thread of control is associated with every thread object. The objects in SPL behave similarly as in Java. A method invocation of an object is *synchronous* and its execution is carried out using a stack. However, for thread objects we introduce a new semantics for method invocation. Specifically, we assume that an invocation of a method of a thread object is *asynchronous* and *atomic*. If a thread of control invokes a method of a thread object, then the method call returns immediately and the call is added as a message to a global message bag. If the thread of control associated with the callee object is not busy processing a message, then it takes out a message (i.e., a call to one of its methods) targeted to it from the global bag and starts executing it atomically and concurrently with other threads. Note that in an execution of a SPL program, several threads can execute concurrently. The atomicity condition requires that for every possible interleaved execution of a SPL program, there exists an equivalent execution with the same overall behavior where the methods of the thread objects are executed serially, that is, the execution of a thread object method is not interleaved with actions of other threads. This particular restriction ensures that the execution of a method of a thread object is not interfered by other threads through shared objects.

$$\begin{aligned}
P &::= \mathit{defn}^*\ (\mathbf{new}\ T).\mathit{md}(c^*)\\
\mathit{defn} &::= \mathbf{class}\ C\ \{\mathit{field}^*\ \mathit{meth1}^*\} \mid \mathbf{thread}\ T\ \{\mathit{field}^*\ \mathit{meth2}^*\}\\
\mathit{field} &::= \mathit{type}\ \mathit{fd}\\
\mathit{meth1} &::= (\mathit{type} \mid \mathbf{void})\ \mathit{md}(\mathit{arg}^*)\{\mathit{local}^*\ \mathit{stmt}^*\}\\
\mathit{meth2} &::= \mathbf{void}\ \mathit{md}(\mathit{arg}^*)\{\mathit{local}^*\ \mathit{stmt}^*\}\\
\mathit{stmt} &::= l\!:\ S;\\
S &::= x = e \mid x.\mathit{fd} = y \mid x.\mathit{md}(y^*) \mid \mathbf{if}\ x\ \mathbf{goto}\ l' \mid \mathbf{return}\ x\\
e &::= \mathbf{new}\ \mathit{type} \mid \mathbf{null} \mid \mathbf{this} \mid c \mid x \mid x.\mathit{fd} \mid x.\mathit{md}(y^*) \mid f(x^*)\\
arg &::= \mathit{type}\ x\\
local &::= \mathit{type}\ y\\
\mathit{type} &::= C \mid T \mid \text{primitive types such as int, float, boolean, etc.}\\
l &::= label\\
x, y &::= \text{variable name}\\
C &::= \text{class name}\\
T &::= \text{thread name}\\
\mathit{fd} &::= \text{a field name}\\
\mathit{md} &::= \text{a method name}\\
f &::= \text{pre-defined functions such as + , -, *, /, etc.}\\
c &::= \text{constants such as 1, 2, true, etc.}
\end{aligned}$$

Fig. 1. SPL Syntax

3.1 Syntax of SPL

The formal syntax of SPL is given in Figure 1. A program in SPL consists of a sequence of definitions of **classes** and **threads** followed by an asynchronous method invocation of a newly created thread object. Observe that the execution of a statement can access at most one shared memory location. This allows us to treat the execution of a statement as an atomic operation. Branching and looping constructs are imitated using the statement **if** x **goto** l, where l is the label of a statement in the method that contains the **if** statement. We assume that a program in SPL is properly typed.

3.2 Semantics of SPL

In the semantics of SPL, we assume that actions of multiple threads can interleave in any way; however, we impose the restriction that the execution of an asynchronous method must be *atomic*. We call this semantics the concurrent semantics of SPL.

The concurrent semantics of SPL is given by augmenting more rules to the standard semantics of Java. Instantaneous snapshot of the execution of a SPL program is called a *configuration*. Formally, a configuration C is a 3-tuple (q, S, M), where

- q is the global state containing the value of every object and thread object currently in use in the program and the program counter of each thread associated with every thread object.
- S is a map from a thread object to an execution stack. The stack for each thread is used in the usual way to execute an asynchronous method sequentially. Note that the invocation of an object method is always synchronous and the method is executed by the caller thread by creating a new stack frame in its stack.
- M is a multi-set or bag of messages. Whenever, a thread invokes a method of a thread object, the target thread object, the method name, and the values of the arguments passed to the method are encoded into a message and placed in the bag. We use $M \cup e$ to represent the multi-set obtained by adding the element e to the multi-set M.

Let $\mathcal{C}$ be set of all configurations. We define a transition relation $C \rightsquigarrow_t^s C'$ (see Figure 2) for the concurrent semantics. Such a relation represents the transition from the configuration C to C' due to the execution of the statement s by the thread t. Henceforth, if t is a thread object, then we will also use t to denote the thread of control associated with the thread t. The transition relations are described abstractly using a number of functions described, informally, below:

- $\textsc{Threads}(q)$ returns the set of thread objects that are created in the execution.
- $\textsc{getNextStatement}(q, t)$ returns the next statement to be executed by the thread t. The function uses the value of the program counter found in q for the thread t to determine the next statement. If the thread t is not executing any asynchronous method, then the function returns $\perp$.
- $\textsc{executeNextStatement}(q, S(t), t)$ executes the next statement of the thread t following the standard sequential Java semantics and returns a pair containing the updated global state q' and the updated map S' in which the stack $S'(t)$ has possibly been modified. The program counter of the thread t is also updated appropriately in the global state q'.
- $\textsc{setNextStatement}(q, S(t), t, t.\mathit{md}(v^*))$, where v denotes a value, creates a stack frame in the stack $S(t)$ to prepare for the invocation of the method *md* and sets the program counter of t in q to the first statement of the method *md* of t. The updated global state q' and the map S' is returned by the function.
- $\textsc{skipNextStatement}(q, t)$ updates the program counter in q of the thread t, such that the thread t skips the execution of the next statement.
- $[\![x]\!]_{S(t)}$ returns the value of the local variable x, which is obtained from the topmost stack frame of the stack $S(t)$.

[Java Semantics]

$$\frac{\begin{array}{c}\exists t \in \textsc{Threads}(q).(s = \textsc{getNextStatement}(q,t)\\ \wedge \quad s \neq \bot \wedge \neg(s = x.md(y^*) \wedge [\![x]\!]_{S(t)} \in \textsc{Threads}(q)))\\ \wedge \quad (q', S') = \textsc{executeNextStatement}(q, S(t), t)\end{array}}{(q,S,M) \leadsto_t^s (q', S', M)}$$

[Consume Message]

$$\frac{\begin{array}{c}\exists t \in \textsc{Threads}(q).(\textsc{getNextStatement}(q,t) = \bot\\ \wedge \quad (q', S') = \textsc{setNextStatement}(q, S(t), t, t.md(v^*)))\end{array}}{(q,S,M \cup \{t.md(v^*)\}) \leadsto_t^\bot (q', S', M)}$$

[Send Message]

$$\frac{\begin{array}{c}\exists t \in \textsc{Threads}(q).(s = \textsc{getNextStatement}(q,t)\\ \wedge \quad (s = x.md(y^*) \wedge [\![x]\!]_{S(t)} \in \textsc{Threads}(q)))\end{array}}{(q,S,M) \leadsto_t^s (\textsc{skipNextStatement}(q,t), S, M \cup \{[\![x]\!]_{S(t)}.md([\![y]\!]^*_{S(t)})\})}$$

Fig. 2. Concurrent Semantics

The initial configuration of a SPL program *defn** (**new** T).*md*(c^*), given by $C_0 = (q_0, S_0, M_0)$, where q_0 contains the thread object, say t, created by the **new** T expression, S_0 maps t to an empty stack, and M_0 contains the only message $t.md(c^*)$. The program counter of t in q_0 is undefined. Thus the Consume Message is the only rule applicable to the initial configuration.

Atomicity Requirement. The concurrent semantics of SPL allows arbitrary interleaving of multiple threads. However, we want to impose the restriction on possible interleavings so that the execution of each asynchronous method is atomic. We next describe this atomicity requirement.

We abstractly represent a finite execution of the form $C_0 \leadsto_{t_1}^{s_1} C_1 \leadsto_{t_2}^{s_2} C_2 \cdots C_{n-1} \leadsto_{t_n}^{s_n} C_n$ of a SPL program following the concurrent semantics by the sequence $\tau = \leadsto_{t_1}^{s_1} \leadsto_{t_2}^{s_2} \cdots \leadsto_{t_n}^{s_n} C_n$. We use $\textsc{MSet}(\tau)$ to represent the multi-set $\{(t_1, s_1), (t_2, s_2), \ldots, (t_n, s_n)\}$. We restrict the set of executions that can be exhibited by a SPL program following the concurrent semantics by imposing the atomicity requirement on asynchronous method executions as follows. If a finite execution $\tau = \leadsto_{t_1}^{s_1} \leadsto_{t_2}^{s_2} \cdots \leadsto_{t_n}^{s_n} (q, S, M)$ be such that $\forall t \in \textsc{Threads}(q).(\textsc{getNextStatement}(q,t) = \bot)$, then τ is said to be a *valid* execution of the program following the concurrent semantics iff the following holds. There exists a finite execution $\tau' = \leadsto_{t'_1}^{s'_1} \leadsto_{t'_2}^{s'_2} \cdots \leadsto_{t'_n}^{s'_n} (q', S', M')$ of the program following the concurrent semantics such that

1. $(q, S, M) = (q', S', M')$,
2. $\text{MS}\textsc{et}(\tau) = \text{MS}\textsc{et}(\tau')$,
3. if for any two elements (t, s) and (t, s') in the $\text{MS}\textsc{et}(\tau)$, $\leadsto_t^s$ appears before $\leadsto_t^{s'}$ in the sequence τ, then $\leadsto_t^s$ also appears before $\leadsto_t^{s'}$ in the sequence τ', and
4. in the sequence τ', all transitions after a $\leadsto_t^\perp$ and before any $\leadsto_{t''}^\perp$ are of the form $\leadsto_{t'}^s$ such that $t = t'$ and $s \neq \perp$.

The above requirement ensures that the execution of an asynchronous method by a thread is atomic. In general, it has been shown that such atomicity requirements for multithreaded programs can be guaranteed statically by using a type system for atomicity [14] or dynamically through rollback [32]. We assume that the language SPL is augmented with an atomicity type system or implemented in a way such that an execution of a program in the language following the concurrent semantics is always valid.

Serialized Semantics. To effectively reason about the behavior of a SPL, we introduce the serialized semantics of SPL and show that for the reasoning purpose we can only consider the serialized semantics of SPL.

Similar to the concurrent semantics, in the serialized semantics, we assume that there is global state q, a global message bag or a multi-set of messages M, and a map S from thread objects to stacks. Then the following happens in a loop. If there is a message (i.e., an asynchronous method call along with values for its arguments) for a thread in the bag, then the thread removes the message from the bag and executes the method in the message. No other thread is allowed to interleave their executions till the execution of the method terminates. During the execution of the method, the executing thread can call asynchronous methods of any thread object. Those calls along with the values for their arguments are placed in the bag as messages. Note that a non-deterministic choice is associated with the picking of a message from the bag.

We define a transition relation $C \longrightarrow_t^s C'$ for the serialized semantics. The rules for transition in the serialized semantics is same as that in the concurrent semantics except for the rule [CONSUME MESSAGE] (see Figure 3). In the serialized semantics, the rule is applicable if none of the threads is executing an asynchronous method and there is a message in the bag. In the concurrent semantics, the rule is applicable if there exists a thread, which is not executing an asynchronous method, and there is a message for the thread in the bag. Note that the atomicity requirement trivially holds in the case of serialized semantics. We represent a finite execution of the form $C_0 \longrightarrow_{t_1}^{s_1} C_1 \longrightarrow_{t_2}^{s_2} C_2 \cdots C_{n-1} \longrightarrow_{t_n}^{s_n} C_n$ following the serialized semantics by the sequence $\longrightarrow_{t_1}^{s_1} \longrightarrow_{t_2}^{s_2} \cdots \longrightarrow_{t_n}^{s_n} C_n$.

[CONSUME MESSAGE]

$$\frac{\begin{array}{c}\forall t \in \textsc{Threads}(q).(\textsc{getNextStatement}(q,t) = \perp) \\ \wedge \quad (q', S') = \textsc{setNextStatement}(q, S(t'), t', t'.md(v^*))\end{array}}{(q, S, M \cup \{t'.md(v^*)\}) \longrightarrow_{t'}^{\perp} (q', S', M)}$$

Fig. 3. Serialized Semantics

Given that a program in SPL always exhibits valid executions following the concurrent semantics, the next result shows that any execution of the program following the concurrent semantics is *equivalent* to an execution of the program following the serialized semantics.

Proposition 3. *For any program execution* $\rightsquigarrow_{t_1}^{s_1} \rightsquigarrow_{t_2}^{s_2} \cdots \rightsquigarrow_{t_n}^{s_n} (q, S, M)$ *where* $\forall t \in \textsc{Threads}(q).\textsc{getNextStatement}(q, t) = \bot$*, there is a serialized execution* $\longrightarrow_{t'_1}^{s'_1} \longrightarrow_{t'_2}^{s'_2} \cdots \longrightarrow_{t'_n}^{s'_n} (q', S', M')$ *such that* $(q, S, M) = (q', S', M')$.

The above result allows us to treat any valid execution of a program in SPL following the concurrent semantics in terms of an equivalent execution following the serialized semantics. Reasoning about a serialized execution is easier because in such an execution we have to consider a sequence of method invocations by different threads, where the execution of each method can be reasoned sequentially. In fact, in the next section we show that reachability of finite programs in SPL is decidable. It is worth mentioning that the reachability of a program in SPL following the concurrent semantics is not decidable if we do not impose the atomicity restriction.

4 Verifying SPL Programs

In this section we consider the problem of verifying SPL programs. Recall that for SPL programs restricted to valid concurrent executions, we observed (in Section 3.2) that reasoning about serialized executions is sufficient when answering questions about global state reachability. Further, during a serialized execution of a SPL program, at any point only one thread executes an asynchronous method up to completion without interleaving with any other thread. This implies that the stack of at most one thread is non-empty at any point in a serialized execution. As a result, we can define the (serialized) semantics using only one stack which is re-used by every active thread.

The verification of serialized SPL programs can proceed by following the familiar methodology of abstracting SPL programs, model checking, checking the validity of a counterexample, and then refining the abstraction if the counterexample if found to be invalid. Using standard predicate abstraction techniques, an SPL program over arbitrary data types can be abstracted into an SPL program all of whose variables are boolean. The steps of checking counterexamples and refining, again can be performed using well-known algorithms. In this section, we therefore focus our attention on model checking finite SPL programs. We first define the formal model of *multi-set pushdown systems* (MPDS) that have finitely many global states, one stack to execute recursive, synchronous method calls, and one message bag to store pending asynchronous method calls. Such MPDSs define the (serialized) semantics of finite SPL programs. We then show that the control state reachability problem for MPDSs is decidable. Finally, we conclude this section by showing that the control state reachability problem has a lower bound of EXPSPACE.

4.1 Multi-set Pushdown Systems

We present the formal definition and semantics of multi-set pushdown systems.

Definition 1. *A multi-set pushdown system (MPDS) is a tuple $\mathcal{A} = (Q, \Gamma, \Delta, q_0, \gamma_0)$, where Q is a finite set of global states, Γ is a finite set of stack and multi-set symbols, $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^* \times \Gamma)$ is the transition relation, $q_0 \in Q$ is the initial state, and $\gamma_0 \in \Gamma$ is the initial method call.*

We let q to range over Q, γ to range over Γ, w to range over Γ^*, M to range over $M_\omega[\Gamma]$. The semantics of an MPDS $\mathcal{A}$ is defined in terms of a transition system as follows. A configuration $\mathcal{C}$ of $\mathcal{A}$ is a tuple $(q, w, M) \in Q \times \Gamma^* \times M_\omega[\Gamma]$. The initial configuration of $\mathcal{A}$ is $(q_0, \epsilon, \{\gamma_0\})$. The transition relation $\longrightarrow$ on configurations is $\longrightarrow_1 \cup \longrightarrow_2$, where $\longrightarrow_1$ and $\longrightarrow_2$ are defined as follows: $(q, w\gamma, M) \longrightarrow_1 (q', ww', M \cup \{\gamma'\})$ if and only if $((q, \gamma), (q', w', \gamma')) \in \Delta$; and $(q, \epsilon, M \cup \{\gamma\}) \longrightarrow_2 (q, \gamma, M)$. Observe that $\longrightarrow_1$ corresponds to the transition rules [JAVA SEMANTICS] and [SEND MESSAGE] and $\longrightarrow_2$ corresponds to the transition rule [CONSUME MESSAGE] in Figure 3. Also note that there is no transition from $(q, \epsilon, \emptyset)$ for any $q \in Q$; therefore, $\mathcal{A}$ halts when it reaches a configuration of the form $(q, \epsilon, \emptyset)$. Finally, $\longrightarrow^*$, $\longrightarrow_1^*$ denote the reflexive, transitive closure of $\longrightarrow$ and $\longrightarrow_1$, respectively.

Definition 2. *A configuration (q, w, M) is said to be* reachable *iff $(q_0, \epsilon, \{\gamma_0\}) \longrightarrow^* (q, w, M)$. A control state q is said to be reachable if for some $w \in \Gamma^*$ and $M \in M_\omega[\Gamma]$, (q, w, M) is reachable.*

4.2 Control State Reachability in MPDSs

We are interested in verifying if a certain global state (or set of global states) of a finite SPL program is reachable. This is the same as checking if a certain control state (or set of control states) is reachable in the MPDS associated with the SPL program. Let us fix an MPDS $\mathcal{A} = (Q, \Gamma, \Delta, q_0, \gamma_0)$. Recall that a control state q is reachable if for some w and M, $(q_0, \epsilon, \{\gamma_0\}) \longrightarrow^* (q, w, M)$. That means for some $q_1, q_2, \ldots q_n$, $\gamma_1, \gamma_2 \ldots \gamma_n$ and $M_1, M_2, \ldots M_n$, we have

$$(q_0, \epsilon, \{\gamma_0\}) \longrightarrow_2 (q_0, \gamma_0, \emptyset) \longrightarrow_1^* (q_1, \epsilon, M_1 \cup \{\gamma_1\}) \longrightarrow_2 (q_1, \gamma_1, M_1) \longrightarrow_1^* (q_2, \epsilon, M_2 \cup \{\gamma_2\})$$
$$\longrightarrow_2 (q_2, \gamma_2, M_2) \cdots \longrightarrow_1^* (q_n, \epsilon, M_n \cup \{\gamma_n\}) \longrightarrow_2 (q_n, \gamma_n, M_n) \longrightarrow_1^* (q, w, M)$$

Thus, the problem of checking whether a control state q is reachable, conveniently breaks up into two parts: for some q', w, γ, M and M', check whether $(q_0, \epsilon, \{\gamma_0\}) \longrightarrow^* (q', \epsilon, M' \cup \{\gamma\})$ and whether $(q', \gamma, M') \longrightarrow_1^* (q, w, M)$. Further observe that $(q', \gamma, \emptyset) \longrightarrow_1^* (q, w, M)$ for some w and M iff $(q', \gamma, M') \longrightarrow_1^* (q, w, M' \cup M)$ for every M'. Hence, we can further simplify our tasks as follows. For some q' and γ, check whether $(q_0, \epsilon, \{\gamma_0\}) \longrightarrow^* (q', \epsilon, M' \cup \{\gamma\})$ for M' and whether $(q', \gamma, \emptyset) \longrightarrow_1^* (q, w, M)$ for some M and w. We will call the first *coverability* problem and the second *control state reachability without context switches* problem. We will treat these problems one by one and show each to be decidable.

Reachability Without Context Switches. We will first consider the problem of checking if for some w, M, $(q', \gamma, \emptyset) \longrightarrow_1^* (q, w, M)$. Observe that since the messages in the bag do not play a role in the transition $\longrightarrow_1$, we can ignore the asynchronous method calls that are generated during a transition in order to decide this problem. Thus, this problem can be reduced to checking reachability in pushdown systems. More formally, consider the pushdown system $\mathcal{P} = (Q', \Gamma', \delta', q'_0, \gamma'_0)$ where $Q' = Q$ the states of the MPDS $\mathcal{A}$, $\Gamma' = \Gamma$, $q'_0 = q'$, $\gamma'_0 = \gamma$, and δ' is defined as follows: $((q_1, \gamma_1), (q_2, w_2)) \in \delta$ iff $((q_1, \gamma_1), (q_2, w_2, \gamma_2)) \in \Delta$ (transition relation of $\mathcal{A}$) for some γ_2. The MPDS $\mathcal{A}$ and the PDS $\mathcal{P}$ are related as follows.

Proposition 4. *A configuration (q_1, w_1) is reachable in $\mathcal{P}$ iff $(q', \gamma, \emptyset) \longrightarrow_1^* (q_1, w_1, M)$ for some M.*

The proof is straightforward and skipped in the interests of space. Hence based on Proposition 4 and Theorem 1, we can conclude that the control state reachability problem, without context switches is decidable in polynomial time for MPDSs.

Coverability. We now study the problem of coverability. Recall that, given a state q' and a stack symbol γ, we need to decide if for some $M' \in M_\omega[\Gamma]$, $(q_0, \epsilon, \{\gamma_0\}) \longrightarrow^* (q', \epsilon, M' \cup \{\gamma\})$. We will introduce a new model of *regular multi-set systems* (RMS), which are slight generalization of *multi-set automata*, and show that the coverability problem can be reduced to a reachability problem on RMS. We will then show that the reachability problem for RMSs is decidable.

Definition 3. *A* regular multi-set system *(RMS) is a tuple $\mathcal{R} = (Q, \Gamma, \delta, q_0, \gamma_0)$, where Q is the set of the states of $\mathcal{R}$, Γ is the multi-set alphabet, $q_0 \in Q$ is the initial state, and $\delta \subseteq ((Q \times \Gamma) \times (Q \times L))$ is the transition relation with $L \subseteq \Gamma^*$ being a regular language. A configuration is the pair (q, M), where $q \in Q$ and $M \in M_\omega[\Gamma]$ and the initial configuration is $(q_0, \{\gamma_0\})$. The semantics of a RMS is given by the transition relation $\hookrightarrow$ over configurations. We say $(q, M \cup \{\gamma\}) \hookrightarrow (q', M')$ iff there is $((q, \gamma), (q', L)) \in \delta$ and $w \in L$ such that $M' = (M \cup \mathbb{M}(w))$.*

Regular multi-set systems are a generalization of multi-set automata, where instead of a transition adding the same multi-set to a bag every time, an RMS transition chooses a multi-set from among a collection described by a regular language and adds to the bag.

We will consider reachability problems for RMSs. A pair (q, γ) is said to be reachable iff there is some M such that $(q_0, \{\gamma_0\}) \hookrightarrow^* (q, M \cup \{\gamma\})$. We will show that the coverability problem of MPDS can be reduced to such a reachability problem. But for that we need to make an important observation about MPDSs.

Proposition 5. *For MPDS $\mathcal{A}$, and any states q_1, q_2 and stack symbol γ_1 define $\mathcal{M}(q_1, q_2, \gamma_1) = \{M \mid (q_1, \gamma_1, \emptyset) \longrightarrow_1^* (q_2, \epsilon, M)\}$. There is a regular language $L(q_1, q_2, \gamma_1)$ such that $\mathbb{M}(L(q_1, q_2, \gamma_1)) = \mathcal{M}(q_1, q_2, \gamma_1)$.*

Proof. Consider the following pushdown automaton $\mathcal{P} = (Q', \Sigma, \Gamma', \delta, q'_0, \gamma'_0, F)$ where $Q' = Q$ the states of $\mathcal{A}$, input alphabet $\Sigma = \Gamma$, stack alphabet $\Gamma' = \Gamma$, initial state $q'_0 = q_1$, initial stack configuration $\gamma'_0 = \gamma_1$, $F = \{q_2\}$, and the transition relation $\delta \subseteq Q \times \Gamma \times \Sigma \times Q \times \Gamma^*$ is defined as follows: $((p_1, \gamma'_1), \gamma'_2, (p_2, w)) \in \delta$ iff $((p_1, \gamma'_1), (p_2, w, \gamma'_2)) \in \Delta$. In other words, $\mathcal{P}$ has a transition on input γ'_2 exactly if the corresponding transition in MPDS $\mathcal{A}$ asynchronously calls γ'_2. Let $L(\mathcal{P})$ be the language accepted by $\mathcal{P}$ simultaneously by empty stack and final state. It is easy to see that $\mathbb{M}(L(\mathcal{P})) = \mathcal{M}(q_1, q_2, \gamma_1)$.

We now recall an important observation due to Parikh [24].

Theorem 2 (Parikh). *For an context-free language L_1 there is a regular language L_2 such that $\mathbb{M}(L_1) = \mathbb{M}(L_2)$. Moreover, given a PDA recognizing L_1 we can effectively construct an automaton for L_2.*

Hence, there is a regular language $L(q_1, q_2, \gamma_1)$ such that $\mathbb{M}(L(q_1, q_2, \gamma_1)) = \mathbb{M}(L(\mathcal{P})) = \mathcal{M}(q_1, q_2, \gamma_1)$. $\square$

Lemma 1. *Given an MPDS $\mathcal{A}$, there is an RMS $\mathcal{R}$ with the same states and multi-set alphabet such that $(q_0, \epsilon, \{\gamma_0\}) \longrightarrow^* (q, \epsilon, M \cup \{\gamma\})$ for any M in the MPDS iff (q, γ) is reachable in $\mathcal{R}$.*

From Lemma 1 we observe that the coverability problem of MPDS is decidable provided checking if (q, γ) is reached in an RMS is decidable. We, therefore, focus on the reachability problem of RMSs. We will show that this problem is decidable by using properties about well-quasi-orderings (wqo) and performing backward reachability as in [12].

For the rest of this section let us fix an RMS $\mathcal{R} = (Q, \Gamma, \delta, q_0, \gamma_0)$. Let us define an ordering $\leq$ over the configurations of a RMS as follows: $(q, M) \leq (q', M')$ iff $q = q'$ and $M \subseteq M'$. An immediate consequence of Dickson's Lemma [9] is the fact that this ordering is a wqo. For a set of configurations S, define $\textsc{pre}(S) = \{(q, M) \,|\, \exists (q', M') \in S. (q, M) \hookrightarrow (q', M')\}$ to be the set of configurations that can reach some configuration in S in one step. Finally, let $\textsc{pre}^*(S) = \bigcup_{i \in \mathbb{N}} \textsc{pre}^i(S)$ be the set of all configurations that can reach some configuration in S in finitely many steps.

Recall that to check if (q, γ) is reachable, we need to see if some configuration in $V = \textsc{cl}(\{(q, \{\gamma\})\})$ is reachable from the initial configuration of the RMS. Hence, we will compute $\textsc{pre}^*(V)$ and check if $(q_0, \{\gamma_0\}) \in \textsc{pre}^*(V)$. Observe that in an RMS, if $(q_1, M_1) \hookrightarrow (q_2, M_2)$ then for every M, $(q_1, M_1 \cup M) \hookrightarrow (q_2, M_2 \cup M)$. Thus, for an upward closed set U, $\textsc{pre}(U)$ is also upward closed. This suggests the following algorithm. Compute progressively the sets U_i, where $U_0 = V$ and $U_{i+1} = \textsc{pre}(U_i) \cup U_i$. The sequence $U_0, U_1, \ldots$ is an increasing sequence of upward closed sets, and so by Proposition 2 we know that this sequence stabilizes in finitely many iterations.

To prove decidability of the reachability problem, all we need to show is that we can compute a representation of U_{i+1}, given a representation of U_i. We can represent an upward closed set U by its minimal elements $\textsc{min}(U)$ which will be finite (by Proposition 1). Thus, we need to describe how to compute $\textsc{min}(U_{i+1})$ from $\textsc{min}(U_i)$.

Consider any upward closed set U and $(q, M) \in \textsc{min}(U)$. For a transition $t_q = ((q', \gamma), (q, L)) \in \delta$ (whose destination is state q) and $w \in L$ define $\textsc{min}(\textsc{pre}^w_{t_q}(U))$ to be $(q', (M \setminus \mathbb{M}(w)) \cup \{\gamma\})$. In other words, $\textsc{min}(\textsc{pre}^w_{t_q}(U))$ is the least configuration that can make a transition using t_q by pushing $\mathbb{M}(w)$ elements into the bag and reach a configuration in U. Let $S = \{\textsc{min}(\textsc{pre}^w_{t_q}(U)) \mid \text{for every } w \in L, (q, M) \in \textsc{min}(U) \text{ and transition } t_q\}$ Our first observation is that S can be represented using regular languages.

Lemma 2. *There are regular languages L_q such that $S = \bigcup_{q \in Q} \{q\} \times \mathbb{M}(L_q)$.*

Finally, we show that given an automaton representation of S, we can compute $\textsc{min}(\textsc{pre}(U))$. From the definition of S it follows that $\textsc{cl}(S) = \textsc{pre}(U)$. Thus

$\textsc{min}(\textsc{pre}(U)) = \textsc{min}(S)$. Our next observation is that given an automaton representation of S, $\textsc{min}(S)$ is computable.

Lemma 3. *Given finite automata A_q for each L_q such that $S = \bigcup_{q \in Q}\{q\} \times \mathbb{M}(L_q)$, $\textsc{min}(S)$ is computable.*

4.3 EXPSPACE Lower Bound

We now show that the control state reachability problem is, in fact, computationally very difficult; we prove the problem is EXPSPACE-hard. The proof relies on ideas for showing the hardness of the reachability problem of Petri Nets due to Lipton [19]. Therefore, we first recall definitions needed to state Lipton's observation, and then sketch how they can be used to prove the lower bound.

Lipton's result can be seen as showing a lower bound for halting problem of special programs called *net programs* [10]. A net program is a finite sequence of labeled commands that manipulate finitely many counter variables. Each statement of a net program is labeled. The basic commands that constitute a net program are as follows: *incrementing* a counter x ($\ell :\ x = x + 1$); *decrementing* a counter x ($\ell :\ x = x - 1$); *unconditional branching* (ℓ : goto ℓ_1); *nondeterministic branching* (ℓ : goto ℓ_1 or goto ℓ_2); *subroutine call* (ℓ : gosub ℓ_1); *return from subroutine* (ℓ : return); and halt (ℓ : halt). So a net program is a sequence of distinctly labeled commands such that the targets of goto and gosub statements are correct labels. Lipton's result applies to *well-structured* net programs, which are programs that can be decomposed into a main program that only calls level 1 subroutines, which in turn only call level 2 subroutines, etc., and the jump commands in a subroutine only have other commands of the same subroutine as target. In terms of such programs Lipton's result can be stated as follows.

Theorem 3 (Lipton [19,10]). *Given a well-structured net program P the problem of checking if some computation ends in the statement halt is EXPSPACE-hard.*

We will prove an EXPSPACE lower bound for the control state reachability of MPDSs by showing that every well-structured net program can be simulated by an MPDS. More precisely, for any well-structured net program P, there is an MPDS $\mathcal{A}(P)$ such that some computation of P halts if and only if a special state q_{halt} is reachable in $\mathcal{A}(P)$. Unfortunately, due to lack of space, we cannot give all the details of the construction of $\mathcal{A}(P)$; instead we will only sketch the main ideas. Corresponding to each labeled statement ℓ : *stmt*, we will have a control state q_ℓ. For each variable x, there will be a multi-set symbol x; the number of such symbols n the bag will denote the current value of x. The stack at all times will have only one symbol, which will be popped at times to remove a message from the multi-set store. We will now sketch the translation of each of the basic commands. The goto statements just involve a change of control state without changing either the stack or the multi-set store. Incrementing x involves making a new asynchronous call to x (i.e., adding x to the multi-set). Decrementing x is a two step process: first we pop the stack (to get to an empty stack) and reach a new control state q'. State q' has the property that if the new method serviced (i.e., removed from multi-set and put on stack) is anything other than x, then it simply makes the same asynchronous call again, pops the stack and goes back to q'; on the other hand if the new method to be serviced is x, it simply new moves to the control state corresponding to the next statement. The idea in simulating subroutine calls is to transfer control to the control state of the subroutine, and then at the same time make an asynchronous call that stores the return address. On a return, we do something similar to the decrement step, to only service the message storing the return address; based on the return address we go to the appropriate new control state. The resulting MPDS has the same order of control states as the net program, and the stack alphabet is also of the same size as the net program. Thus, based on all these observations, we have the following theorem.

Theorem 4. *The control state reachability problem for MPDSs EXPSPACE-hard.*

Acknowledgment

We would like to thank Gul Agha, Chandrasekhar Boyapati, Cormac Flanagan, Ranjit Jhala, and Shaz Quadeer for providing valuable comments on this work. This work is supported in part by the ONR Grant N00014-02-1-0715, the NSF Grants NSF CNS 05-09321, NSF CCF 04-29639, and NSF CAREER 04-48178.

References

1. J.-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. pages 111–174, 1997.
2. T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *LNCS*, pages 260–264, 2001.
3. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*, volume 3821 of *LNCS*, 2005.
4. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Principles of Programming Languages (POPL'03)*, 2003.
5. A. Bouajjani, M. Mueller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. 16th Intern. Conf. on Concurrency Theory (CONCUR'05)*, volume 3653 of *LNCS*, 2005.
6. D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106:61–86, 1992.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, 2004.
8. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 331–346, 2003.
9. L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with r distinct prime factors. *American Journal of Mathematics*, 35:413–422, 1913.
10. J. Esparza. Decidability and complexity of Petri net problems — An introduction. In *Lectures on Pteri Nets I: Basic Models. Adavnaces in Petri Nets*, number 1491 in Lecture Notes in Computer Science, pages 374–428. 1998.
11. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Principles of Programming Languages (POPL'00)*, pages 1–11, 2000.
12. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001.

13. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking push-down systems. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *Electronic Notes in Theor. Comp. Sci.* Elsevier, 1997.
14. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'03)*, 2003.
15. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. of the 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.
17. A. Holub. *Taming Java Threads*. APress, 2000.
18. J. B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory: Series A*, 13(3):297–305, 1972.
19. R. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.
20. D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 274(1–2):89–115, 2002.
21. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, Technical University Munich, 1998.
22. F. Moller. Infinite results. In *Proceedings of the Conference on Concurrency Theory*, pages 195–216, 1996.
23. M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theoretical Computer Science*, 311(325–388), 2004.
24. R. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.
25. S. Qadeer, S. Rajamani, , and J. Rehof. Procedure summaries for model checking multithreaded software. In *Principles of Programming Languages (POPL'04)*, 2004.
26. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 93–107, 2005.
27. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, 2004.
28. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.
29. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83, 1997.
30. H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. In *European Symposium on Programming (ESOP'00)*, volume 1782 of *LNCS*, 2000.
31. K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. Technical Report UIUCDCS-R-2006-2683, UIUC, 2006.
32. A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *LNCS*, pages 519–542. Springer, 2004.