Model Checking Java Using Pushdown Systems

Jan Obdržálek

LFCS, Division of Informatics
The University of Edinburgh
J.Obdrzalek@ed.ac.uk

Abstract

In recent years, model checking algorithms for the verification of infinite-state systems were introduced. We evaluate the possibility of using the algorithms for pushdown systems and various modal logics of [3] for verification of Java programs. It turns out that pushdown systems are particularly suitable for modeling the control flow of sequential Java programs, including exceptions (which are too often not supported by model checking tools). The process of mechanical abstract model generation is described, with emphasis on supporting Java exception handling mechanism, which is the major new contribution of this paper. We also show some possible uses of our approach and present a survey of the related work.

1 Introduction

At present, mainly finite-state tools such as SPIN[11] or SMV[15] are used to model check software systems. On the other hand, programs written in commonly used languages are sources of infinite-state behaviour. This is because of the presence of (possibly) infinite data structures and/or infinite control structures such as the call stack. The problem of infinite data is tackled by using well known techniques such as data abstraction and slicing. For Java, these were recently successfully applied in the Bandera toolset[5] and Java PathFinder[22].

However, the problem of infinite control arising from recursion is rarely considered and method inlining is used to obtain finite-state program models. This unfortunately prevents us from checking important properties based on the call stack behaviour. In the last few years, several model checking algorithms for infinite-state systems appeared. For programs with recursion, the algorithms for model checking pushdown systems (PDS) using the LTL and modal μ-calculus logics are of a great interest, since these can accurately describe program execution.

Pushdown systems themselves seem to be a natural choice for modeling the method call stack and interprocedural program behaviour. Contribution of this paper is in taking (sequential) Java as the language in question and showing, that PDS can be really useful model of this language. The most interesting to us is the observation that PDS can accurately model the Java exception flow (both intra- and interprocedural), even in the presence of multilevel `break`/`continue` construct. Since exceptions are being heavily used in Java programs, this feature shows PDS to be a really useful model. Combined with the previously mentioned techniques such as abstraction, this can lead to qualitatively better model checking tools.

The organization of the rest of our paper is as follows: We start by presenting a brief overview of PDS and related results. In the next section we show how to model various features of Java program using PDS. The main result of this paper, modeling the exception flow of Java programs, is presented in a separate section. We continue by sketching some of possible uses of PDS model checking in Java verification. Finally, we present an overview of related work and end by a conclusion.

2 Pushdown Systems and Model Checking

Pushdown systems (PDS) are basically pushdown automata seen from a different point of view: we are not interested in the languages they recognize, but in the transition systems they generate. These are infinite (in our case unlabelled) transition systems, having states of the form $\langle$control location, stack contents$\rangle$.

Definition 1 *A* pushdown system *(or PDS for short) is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$*, where:*

- P *is a finite set of* control locations*,*
- Γ *is a finite* stack alphabet*, and*
- $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ *is a finite set of* transition rules*.*

A configuration *is a pair* $\langle p, w \rangle$*, where* $p \in P$ *is a control location and* $w \in \Gamma^*$ *is a* stack content*.*

Every PDS $\mathcal{P}$ *defines an associated transition system* $\mathcal{T} = (\mathcal{S}, \rightarrow)$*, where* $\mathcal{S}$ *is a set of all configurations of* $\mathcal{P}$ *and* $\rightarrow$ *is the obvious extension of* Δ *to configurations. I.e.* $\langle p, \gamma w \rangle \rightarrow \langle q, vw \rangle$ *iff* $((p, \gamma), (q, v)) \in \Delta$*.*

2.1 Model Checking Results

Given a formula φ of some temporal logic and a PDS $\mathcal{P}$, the model checking problem consists of deciding whether a configuration c of $\mathcal{P}$ violates φ. Efficient algorithms for model checking PDS using the LTL and modal μ-calculus logics were given in [3, 7, 8].

Two model checkers for PDS appeared in the last years [7, 17]. The first of them, named Moped[18], uses the LTL logic and it is based on the algorithm given in [7]. The algorithm used is polynomial in the number of control states of a pushdown system and singly exponential in the size of a formula. Available atomic propositions are of the form "*the stack symbol s is currently on top of the stack*", whereas the algorithm given in [7] allows also the control locations to be taken into account. A recent work [8] presents a modified LTL model checking algorithm allowing regular predicates on the stack contents.

The second tool, called Alfred [17], is a model checker for alternation-free modal μ-calculus and PDS. It uses the algorithm described in [3] and arbitrary regular predicates on the stack contents are allowed. The algorithm is singly exponential in the number of states of a pushdown system and in the size of a formula.

3 Translation of Java into PDS

Pushdown systems are especially suitable for modeling sequential programs with recursive procedures. The model we use establishes a relation between the control points of a program and corresponding configurations of the PDS generated. This is achieved by using the PDS stack to model the call stack of a program. The following sections show only basic principles, while a detailed description can be found in [16].

3.1 The Java Pushdown Model

This section briefly describes the core basics of encoding recursive programs (so not only Java) by PDS. The discussion of fairly complex exception handling is deferred to Section 4.

The PDS model is built in two steps. Firstly, a control flow graph is built for every method declared in the program. Nodes of the flow graph represent the control points of the method and edges are labelled with the corresponding statements. Since we abstract from the values of variables, conditions (e.g. the `if-then-else` construct) are modeled by a nondeterministic choice. Secondly, every method call in the flow graph is coupled with the start node of the called method.

The pushdown system construction works as follows: Assume we have a system of control flow graphs where N is the set of all the control points. We construct a (single state) PDS $\mathcal{P} = (\{p\}, N, \hookrightarrow)$ with N as its stack alphabet. In this system, a configuration $\langle p, nw \rangle$ represents the program being at the control point n, where w are return addresses of the calling procedures.

More formally, the transition relation $\hookrightarrow$ is defined by the following rules:

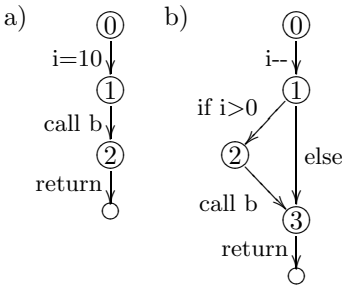
- $\langle p, n \rangle \hookrightarrow \langle p, n' \rangle$ iff there is an (n, n') edge in the corresponding control flow graph, which is not a method call.

- $\langle p, n\rangle \hookrightarrow \langle p, mn'\rangle$ iff (n, n') is a method call edge and m is an entry point of the method being called.
- $\langle p, n\rangle \hookrightarrow \langle p, \varepsilon\rangle$ iff the edge leaving n contains the `return` statement.

The process of building a PDS from a Java source is depicted in Fig. 1. Note that in our simple example we have only one control location, as we do not model any data or exception flow.

```
void a(){
  int i=10;
  b(i);
}
```

```
void b(int i){
  i--;
  if (i>0)
    b(i);
}
```



$\langle p, a_0\rangle \hookrightarrow \langle p, a_1\rangle$
$\langle p, a_1\rangle \hookrightarrow \langle p, b_0\ a_2\rangle$
$\langle p, a_2\rangle \hookrightarrow \langle p, \varepsilon\rangle$

$\langle p, b_0\rangle \hookrightarrow \langle p, b_1\rangle$
$\langle p, b_1\rangle \hookrightarrow \langle p, b_2\rangle$
$\langle p, b_1\rangle \hookrightarrow \langle p, b_3\rangle$
$\langle p, b_2\rangle \hookrightarrow \langle p, b_0\ b_3\rangle$
$\langle p, b_3\rangle \hookrightarrow \langle p, \varepsilon\rangle$

Figure 1: Java program with the corresponding control flow graph and pushdown system.

3.2 Virtual Method Call Resolution

Since Java is an object-oriented language, we must be aware of the presence of virtual method calls. In this case, since *late binding* is always used in Java, we can not statically determine exactly which method is going to be invoked (since a reference to type `T` through which a method is invoked can point to an object of some subtype of `T`). Instead we compute the sets of all possibly invoked methods and allow a nondeterministic choice at the point where a method is called.

Of course, virtual method call resolution is not a new problem and was, in the context of data-flow analysis, extensively studied for a variety of object-oriented languages. We use the well-known technique called *class hierarchy analysis (CHA)* [6], which is based on the following observation: If `a` is a reference to an object of type `T`, then it can either point to an object of class `T` or to an object of a class type derived from `T`. However, if a method `T.m()` is not redefined in any class type derived from `T`, then `a.m()` always calls the method `T.m()` (which is either defined in `T`, or inherited from its superclass). It was shown[6] that in practice CHA substantially decreases the number of methods which can be invoked by a given method call.

It should be noted that class hierarchy analysis is not the only technique solving the virtual method call resolution problem. For example, a recent work [20] shows a more sophisticated approach to this problem in the context of the Java programming language.

3.3 Modeling Data

Our system is able to capture only the control flow – but we are usually interested in data as well. In PDS, there is an obvious way of handling data: Locations can be extended (using the product construction) to hold global data, whereas local data (as e.g. local variables) are to be kept on stack (additional information is stored in stack symbols). However, this leads to an increase in the complexity of model checking. Having m bits of global and n bits of global data, the time complexity increases by 2^{3m+n} in the worst case (using the algorithm of [7]). Unfortunately, the space complexity increases by a similar number, which causes implementation problems.

One way to tackle this problem is to use a symbolic representation of states. Esparza and Schwoon [9] have chosen Binary Decision Diagrams (BDDs) and present the notion of *symbolic pushdown system*, which extends our definition of PDS. They show that the use of BDDs can lead to exponentially smaller models.

Formally, a symbolic PDS it is a triple $\mathcal{P}_S = (P \times G, \Gamma \times L, \Delta_S)$, where G and L are set of global and local values and Δ_S is a set of *symbolic transition rules*, which are of the form $\langle p, \gamma\rangle \xhookrightarrow{R} \langle p', \gamma_1 \dots \gamma_n\rangle$, where $R \subseteq (G \times L) \times (G \times L^n)$ is a relation. Of course it is necessary for R to have an efficient representation. A symbolic pushdown system corresponds to a normal pushdown system $(P \times G, \Gamma \times L, \Delta)$ in the sense that a symbolic rule $\langle p, \gamma\rangle \xhookrightarrow{R} \langle p', \gamma_1 \dots \gamma_n\rangle$ denotes a set of transition as follows:

If $(g, l, g', l_1, \dots, l_n) \in R$, then $\langle (p, g), (\gamma, l)\rangle \hookrightarrow \langle (p', g'), (\gamma_1, l_1) \dots (\gamma_n, l_n)\rangle$.

The results presented in [9] for verification of the notorious QuickSort algorithm are pretty encouraging and deserve a closer look.

4 Modeling Exception Flow

Next we extend our model to support Java exception handling mechanism. This is quite involved (see [14]), especially dealing with the exceptions propagating through method calls. Our solution is based on using both stack and control locations to hold the necessary information: With every control location we associate the information whether there is an exception being propagated and if so, of which type.

The solution here may seem to be too complicated to do the work, mainly because of storing information on stack where possible. The reason for this is to keep the number of stack symbols and control locations as low as possible to decrease the time and space needed to model check the system. Whether using the more obvious representation together with the symbolic techniques presented in Section 3.3 is a better way to do this, we do not know.

Assuming we have a PDS representation $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ of a Java program, we will show how to get a new PDS $\mathcal{P}' = (P', \Gamma', \hookrightarrow')$ which truthfully models Java exceptions. The basic idea is this: Let E be the set of all exception types thrown somewhere in the program. We define a new set of control locations $P' = \{(p, e) \mid p \in P \ \wedge \ e \in (E \cup \{0\})\}$. The second component of a (p, e) pair is an *exception flag*. The exception flag e means that an exception of type e was thrown, and 0 that no exception is currently being propagated. This implies that every transition $\langle p, \alpha\rangle \hookrightarrow \langle q, \beta\rangle$ is substituted by the $\langle (p, 0), \alpha\rangle \hookrightarrow' \langle (q, 0), \beta\rangle$ transition. The transitions associated with exception flow will be presented in the following section. For simplicity, from now on we will only show the exception flag component of the state.

To handle the `finally` construct, we also have to extend our set Γ of stack symbols. The stack will hold information about the `try` block being processed and the mode in which this `finally` was called. (On the exceptional exit we must continue propagating the exception after executing the `finally` block.) Therefore we a) introduce a new special stack symbol Ω_x associated with every `try` block x, and b) we require that $E \subseteq \Gamma'$.

We make the following assumptions:

- There is a single exit terminal node for the `try` and `catch` and `finally` CFGs. This could be easily satisfied if there are no `break`, `continue` and `exit` statements present. Handling of these statements will be described separately.
- There is a `finally` clause associated with every `try` block. If this is not the case, we can simply add a void `finally` clause with no effect on the program semantics. (*Notation:* The first node of a `finally` block is called f in the rest of the section).

4.1 Modeling `try-catch`

The `throw` statement Let t be the CFG node corresponding to a `throw` statement, and e the type of the exception being thrown. (Since the syntax in fact is `throw` *Expression*;, there may be

possibly more than one exception type thrown at this point. In such a case we introduce additional nondeterminism as in the case of method calls.) For brevity we present only the simple instance. There are two possible cases:

- e is caught by the ith `catch` clause (with c_i as a first node). In this case we add a transition $\langle 0, t\rangle \hookrightarrow \langle 0, c_i\rangle$. Notice that the exception flag is cleared after the transition – since the exception have already been caught. (*Remark:* Exceptions thrown in a `catch` block are handled as the next case.)
- e is not caught. Finally block is then going to be executed in exceptional mode, the reason being the exception of type e. The corresponding transition is $\langle 0, t\rangle \hookrightarrow \langle 0, f\, e\, \Omega_x\rangle$.

Normal completion If the block completes normally, then the control is in node t_ε and the `finally` block is to be called in normal mode. The corresponding transition is $\langle 0, t_\varepsilon\rangle \hookrightarrow \langle 0, f\, 0\, \Omega_x\rangle$. (*Remark:* The same holds for normal `catch` block exit.)

4.2 Modeling `finally`

As we could see earlier, the top of the stack inside the `finally` block is of the shape $f\, e\, \Omega_x \ldots$ if `finally` was called in exceptional mode and $f\, 0\, \Omega_x \ldots$ otherwise. When the control is about to leave the `finally` block, it simply pops the top stack symbol. The next behaviour depends on both the state and stack symbol. There are two possibilities:

Exceptional exit An exception was raised in the `try-catch-finally` block and not handled there. Then either:

a) **the `finally` block was called in exceptional mode and completed normally**
Then exception flag is 0 and e is on top of the stack. We introduce a set of transitions $T_1 = \{\langle 0, e\rangle \hookrightarrow \langle e, \varepsilon\rangle \mid e \in E\}$. These transitions are shared by all the methods.

b) **the `finally` block was called in exceptional mode and completed abruptly**
Then exception flag is e and 0 is on top of the stack. As in previous state we introduce a set of transitions $T_2 = \{\langle e, 0\rangle \hookrightarrow \langle e, \varepsilon\rangle \mid e \in E\}$, also common to all the methods.

c) **an exception was thrown in both the `try` and `finally` blocks**
Both the exception flag and top of the stack contain an exception type. In this case, the more recent reason is used. This implies a family of transitions $T_3 = \{\langle e_1, e_2\rangle \hookrightarrow \langle e_1, \varepsilon\rangle \mid e_1, e_2 \in E\}$.

The reader can see that in all three cases we end up in a configuration $\langle e, \Omega_x \ldots\rangle$. The behaviour now depends on whether there is a lexically enclosing `try` block.

- The `try` block is present. Then transitions are added as if the exception of type e was thrown by the `throw` statement in the enclosing `try` block.
- The `try` block is not present. Then the behaviour is the same as if the exception of type e was thrown by the `throw` statement not enclosed by a `try` block (this is described below).

Normal completion Firstly, we must pop the stack using the $\langle 0, 0\rangle \hookrightarrow \langle 0, \varepsilon\rangle$ transition. Then we continue with the node (say n) immediately following the `try-catch-finally` construct. The corresponding transition is $\langle 0, \Omega_x\rangle \hookrightarrow \langle 0, n\rangle$.

4.3 Modeling Exception Propagation Through Method Calls

As was mentioned earlier, exception flag is used to propagate exception through method calls.

`throw` statement outside a `try` block When such a `throw` is reached the top stack symbol is removed from the stack and the exception is left to be handled by the calling method. More formally, let i be the CFG node corresponding to a `throw` statement, and e the type of the exception being thrown. Then we add a transition $\langle 0, i\rangle \hookrightarrow \langle e, \varepsilon\rangle$.

Modeling a method call When the called method returns, we must distinguish, whether it completed normally or abruptly. For this reason, we introduce a whole set of new transitions for every method call site. These will transfer the control to the corresponding `catch` /`finally` clauses when necessary.

More formally, let n' be node immediately following the method call (this node is on top of the stack when the called method terminates). For every exception type $e \in E$ we introduce a transition $\langle e, n'\rangle \hookrightarrow \sigma$, where σ is a target of a transition corresponding to the `throw e` statement, would such transition appear at n'.

4.4 Handling of the `return`, `break`, and `continue` Statements

When handling exceptions, we must be aware of the following fact. If control leaves the `try` block as a result of executing one of the `return`, `break` or `continue` statements (also referred to as *control transfer statements*, then the associated `finally` block must be executed before the control transfers to a new destination. Moreover, the above mentioned statements can also appear in the `catch` and `finally` blocks.

Handling of the control transfer statements is very similar to handling exceptions and can be done by introducing a new exception type for every such a statement. New transitions are added as for the "normal" exception types, with these modifications:

- When a Ω_x symbol is on a top of the stack, the transition is made according to the semantics of the present exception type.
- Exception types corresponding to the control transfer statements do not propagate through method calls.

5 Uses of PDS Model

The PDS model of a Java program can be used to check arbitrary properties which can be expressed in one of the logics supported by available PDS model checkers. The following example, taken from [7], shows how a complex call dependency between methods can be verified. Look at the program in Fig. 2. This is a program for plotting random bar graphs using the commands `goUp`, `goRight`, and `goDown`. One of the correctness properties is the requirement, that an upward movement will not be immediately followed by a downward movement and vice versa.

```
static void m(){
  double d = Math.random();
  if (d < 0.66) {
    s(); goRight();
    if (d < 0.33) m();
  } else {
    goUp(); m(); goDown();
  }
}
```

```
static void s() {
  if (Math.random() < 0.5)
    return;
  goUp();  m();  goDown();
}

public static void main(String args[]) {
  s();
}
```

Figure 2: The graph plotting program

The desired property can now be specified using the following formula:

$$\begin{aligned} \varphi &= \mathbf{G}(goUp \rightarrow (\neg goDown\ \mathbf{U}\ goRight)) \\ &\wedge \mathbf{G}(goDown \rightarrow (\neg goUp\ \mathbf{U}\ goRight)) \end{aligned}$$

where the atomic predicates *goUp*, *goRight*, and *goDown* correspond to the `goUp()`, `goRight()`, and `goDown()` methods being called (i.e. their corresponding entry points are on top of the call stack). This formula evaluates to true in our system confirming that the correctness assumption is satisfied.

5.1 Using State to Hold Information

In addition to talking about program locations, we can also use states to hold some global information. This allows us to speak about more complex program properties without using overwhelmingly complicated formula.

This approach can be used e.g. for verifying resource constrains. This is important in small systems with bounded resources, such as space systems[4] or smart cards. The state component in this case holds a bounded counter, which is increased(decreased) when a resource is allocated(deallocated). The problem is then reduced to checking whether any configuration with counter bigger than maximum value is reachable, which can be easily done using the **F** operator.

5.2 Exception Flow Analysis

Static analysis of Java exception flow can be used during program development to check that exceptions are handled accurately by the program. However, this task becomes very complicated in the presence of recursion, since exceptions can propagate through method calls. Various methods used for static analysis sometimes restrict the exception structure (i.e. by not supporting the cycles in call graph). A general algorithm to be used in the context of classical static analysis was recently presented in [19].

On the other hand, since the PDS model of Java program takes the flow of exceptions into account, it can be readily use for analysis. This allows the programmer to spot potential sources of errors in his programs. Using a single LTL formula, it is easy to check whether some exception is always caught by the program or that there is not the possibility of an exception being thrown and not immediately handled in some critical part of program.

5.3 Application to Java Security

The Java Development Kit (JDK) 1.2 introduced a completely new security model, which is based on the notion of *protection domains*. Every Java class is assigned to some protection domain, and every protection domain is associated with a set of *permissions*. Permissions are verified using the `AccessController` class from the standard library. The `CheckPermission` method of this class checks whether a given permission is granted and throws an exception if not.

For a permission to be granted, all methods on the call stack must have this permission granted OR a caller whose domain is granted the said permission has been marked as *privileged* and all of the methods subsequently called by this caller have the permission. In other words, methods on the call stack are checked in a top-down manner, until either the stack bottom or a privileged caller is reached. A caller is marked as privileged when it calls the `doPrivileged` method [1]. This method takes as an argument an object of the `PrivilegedAction` class, whose only method `run()` contains the code to be made privileged. In all the examples we have seen, this is done using an anonymous inner class. The privileged call can therefore be identified statically.

[1] In the first releases of JDK 1.2, the `beginPrivileged/endPrivileged` method pair was used instead of the `doPrivileged` method to designate privileged code.

Formalisation in our model
Since the security in the terms of possible call stack contents, we can use our model in a straightforward manner. Our work was inspired by the paper [13] which deals with the same problem.

Having built a PDS representation of a Java program, we can identify the set of permissions associated with every stack symbol (which in turn corresponds to some control point of the program). As we said earlier, we also can statically identify the set of stack symbols which have been marked privileged. Having this information, we can describe the interesting call stack structures (e.g. those violating some security property) using regular predicates on the stack contents.

6 Related Work

6.1 Model Checking Java Programs

In recent years, several attempts have been made to automatically model check Java programs. The first generation tools such as JCAT [12] and JAVA PATHFINDER [10] translated a Java program directly into a relatively expressive PROMELA language, the input language of the SPIN [11] model checker. Both tools support concurrent thread execution and allow user to annotate programs with various assertions. The weakness of these tools (in addition to the discussed finite-state limitations) is their inability to significantly compress the PROMELA model based on the property to be checked, so model checking for nontrivial programs easily becomes intractable.

The BANDERA tool-set [5], which utilises the SOOT framework, overcomes some of these limitations and it seems to be the most sophisticated tool at the moment. BANDERA employs sophisticated abstraction and slicing techniques, which restrict variable domains and make the resulting model significantly smaller. The generated model is encoded in an intermediate language representation, which is then converted to the input language of some available model checker (e.g. SPIN [11] or SMV [15]). As far as we know, exceptions are not supported yet.

JPF(2) - the second generation of JAVA PATHFINDER[22] - is another very interesting model checker. JPF is an explicit state model checker (similar to SPIN), which operates directly on Java bytecode and implements its own JVM. It incorporates several search algorithms, as well as sophisticated abstraction techniques. Moreover, slicing algorithm of BANDERA is used for reducing state space. Full integration with BANDERA is under way.

6.2 Verification of Software Using Infinite-state Systems

In the area of model checking with infinite-state systems, there are only few existing implementations of so far discovered verification techniques for checking programs written in widely used languages as C or Java. The SLAM toolkit [2] supports the verification of C programs and has been used to validate security properties of Windows NT device drivers. In SLAM, C programs are translated into a model called *boolean programs*, which are basically C programs where all variables have boolean type. The process of model abstraction is highly automated and models are automatically refined when necessary. A symbolic model checker BEBOP [1], which is used to verify boolean programs, represents sets of states implicitly using BDDs and uses an interprocedural data-flow analysis algorithm to solve the reachability problem.

In [9], another model checker for boolean programs is presented. In contrast to BEBOP, this model checker can deal with liveness and fairness properties, since arbitrary LTL properties are allowed. The model checker works with symbolic pushdown systems (SPDS), a compact representation of the pushdown systems studied in [7]. As it is further shown in [9], this approach leads to efficiency advantages over the BEBOP tool. The MOPED tool[18] was extended to handle the boolean programs of SLAM.

6.3 Bytecode Verification

Instead of verifying a program given its source code, we can start with the compiled bytecode. This makes the program analysis much simpler, since bytecode has quite a restricted set of basic

statements. Even in the case the bytecode has been optimised, there is still a Java source code line number associated with every statement, allowing the generated error trace to be matched with statements of a program source code. The advantage of this approach is that we can verify functionality of the whole program even in the case that for some of its parts (libraries etc.) the source code is not available.

Since bytecode is a stack based code, the analysis is not quite straightforward. This obstacle can be overcome using the SOOT compiler framework developed by the Sable group at the University of McGill [21]. Using SOOT, Java programs are transformed into an intermediate JIMPLE representation (one of intermediate representations available in SOOT). JIMPLE is a typed, 3-address representation of bytecode, where stack is eliminated and replaced by additional local variables. Another advantage is that we can employ sophisticated techniques for virtual method call resolution, which were implemented as a part of the SOOT framework [20].

7 Conclusion and Future Work

As we have shown, pushdown systems can be advantageously used for verifying sequential programs written in modern, high-level programming languages. PDS allows us to model the call stack behaviour, taking into account even such complicated constructions as is Java exception handling mechanism together with multilevel `break/continue`. The presence of really effective model checking algorithms for powerful logics makes this even more interesting.

But for our approach to be widely used, there is much more work to be done. In addition to developing the theory especially in handling the data flow, the other important step is to produce a verification tool based on the principles shown above. Such a tool should be able to work both on the source code and bytecode level, for the reasons mentioned earlier. A very promising option is to connect such a tool to the BANDERA toolkit, which would supply it also with a range of a useful slicing and abstraction algorithms. Moreover, the JIMPLE intermediate code used by BANDERA is easier to handle then the Java source. Performance tests on a large set of sample programs should be run and the obtained results than evaluated to create a verification tool which could be used by software engineers.

8 Acknowledgement

I'm very grateful and would like to thank the anonymous referees for their helpful comments on the earlier draft of the paper.

References

[1] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN' 00: Model Checking of Software*, 2000.

[2] T. Ball and S. K. Rajamani. Checking temporal properties of software with boolean programs. In *Workshop on Advances in Verification (with CAV 2000)*, July 2000.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag.

[4] Edmund Clarke, David Garlan, Bruce Krogh, Reid Simmons, and Jeannette Wing. Formal verification of autonomous systems NASA intelligent systems program. Available from: `http://www-2.cs.cmu.edu/~rareglitch/proposals/formal-v5.pdf`.

[5] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer Verlag, 1995.

[7] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV: International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer-Verlag, 2000.

[8] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 306–339. Springer-Verlag, 2001.

[9] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV: International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer-Verlag, 2001.

[10] K. Havelund and J. U. Skakkebæk. Applying model checking in Java verification. In *Proceedings of the 6^{th} SPIN Workshop*, pages 216–231, Toulouse, France, 1999.

[11] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[12] R. Iosif, C. Demartini, and R. Sisto. Modeling and validation of Java multithreading applications using SPIN. In *Proceedings of the 4^{th} SPIN Workshop*, Paris, France, 1998.

[13] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. Technical report, IRISA, Rennes, 1999.

[14] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[15] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[16] Jan Obdržálek. Formal verification of sequential systems with infinitely many states. Master's thesis, FI MU Brno, 2001. Available from: `http://www.fi.muni.cz/~xobdrzal/`.

[17] Daniel Polanský. Implementation of the modelchecker for pushdown systems and alternation-free mu-calculus. Master's thesis, FI MU Brno, 2000. Available from: `http://www.fi.muni.cz/paradise/papers/xpolansk.ps`.

[18] S. Schwoon. Moped - a model-checker for pushdown systems. Available from `http://wwwbrauer.in.tum.de/~schwoon/moped/`.

[19] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *Software Engineering*, 26(9):849–871, 2000.

[20] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, volume 35.10 of *ACM Sigplan Notices*, pages 264–280. ACM Press, 2000.

[21] R. Vallée-Rai, Phong Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON '99*, 1999.

[22] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. In *Workshop on Advances in Verification (with CAV 2000)*, July 2000. Available from: `http://ase.arc.nasa.gov/havelund/Publications/jpf2-postcav.ps`.