policies that are always respected. This static analysis can be split in two phases:

- in the first phase, we extract from the bytecode a control flow graph, and we transform it into a history expression [4]. This is a sort of context-free grammar, the language of which over-approximates all the possible traces of events that the analysed program can generate at run-time.
- in the second phase, we reduce the infinite-state system given by the history expression to an equivalent finite one, and check it against the usage policies mentioned by the sandboxes used in the program. This is done through a model-checker. Only the policies that do not pass model checking need to be enforced at run-time. This phase has been implemented by our LocUsT tool, which runs in polynomial time in the size of the extracted history expression. Further details about this phase can be found in [4,6].

For the third point, we are developing an Eclipse plugin that combines the previous items into a programming environment, with facilities for writing policies, sandboxing code, and for running the static analyses to discover which policies can be disregarded by the security monitor. The LocUsT model checker, a prototype of this first analysis phase, and a prototype of the Eclipse plugin are distributed along with the Jalapa sources through the SourceForge Subversion repository.

Acknowledgments. Research partially supported by EU-FETPI Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers), by PRIN Project SOFT (Tecniche Formali Orientate alla Sicurezza), and by EU project IST-033817 GridTrust (Trust and Security for Next Generation Grids).

References

The Jalapa runtime environment. The core of the enforcement mechanism is
the method PolicyPool.check() that, for each active policy, tracks the states of all
the needed usage automata. The state of the monitor is a mapping from policies to
sets of pairs \((\mathbf{O}_1, \ldots, \mathbf{O}_k, Q)\), where \((\mathbf{O}_1, \ldots, \mathbf{O}_k)\) is a tuple of weak references to the
objects that substitute the formal parameters of the usage automaton, and \(Q\) is the
current state of the usage automaton. Dummy instantiations are also maintained,
to be concretized when new objects are discovered in the execution trace. When an
object is garbage-collected, its occurrences in the mechanism state are reverted to
dummies. If no usage automaton reaches an offending state, the intercepted method
call is forwarded to the actual target; otherwise, a security exception is thrown.

Supporting artifacts. Jalapa is an open-source project. The sources are avail-
able through a Subversion repository at SourceForge [10]. Some further supporting
material is accessible through the project Web page:

- the Jalapa Tutorial, that provides programmers with a step-by-step guide for
  securing Java programs with local usage policies.
- a repository of example programs and policies, including a prototype imple-
  mentation of a secure bulletin board system.
- the manual page of policies, that defines their syntax and semantics.
- the manual page of the Jalapa rewriter, that defines its command-line syntax.

3 Discussion: present and future challenges

The Jalapa project started as an applicative branch of more foundational work on
history-based access control [3,4,5]. Porting this theoretical machinery to a concrete
setting like Java posed several issues. While our original goals have been achieved to
a fair degree by the current release of Jalapa, there is room for future improvements.
We devise three main research directions: (1) increasing the expressive power of
usage policies, (2) reducing the run-time overhead of the enforcement mechanism,
and (3) developing programming tools and methodologies to facilitate writing secure
programs with Jalapa.

For the first point, although our usage policies are quite general, they do not
cover all the possible real-world scenarios. We would like to require e.g. that a given
low-level method (e.g. a write-file) can only be invoked within the scope of some
high-level method that securely manages the low-level one. This is the case e.g. of
a change-password method that calls write-file to update passwords. The challenge
is to improve the expressive power of usage policies, while keeping them clean and
formally sound. A promising solution seems that of introducing aliases of the form
\(ev := C_1.m_1(...) \{ C_2.m_2(...) \}\), meaning that the event \(ev\) is fired whenever
the method \(m_1\) of class \(C_1\) is invoked within the scope of the method \(m_2\) of class \(C_2\).

Another improvement would be to allow policies to mention the values
returned by methods. This can be done by generating “return” events, exposing these values.

For the second point, we are currently developing a static analyser for Java byte-
code, to detect those policies that are always respected in all the possible executions
of the application. The run-time enforcement can then be optimized, by discarding
the wrappers, and the associated execution monitoring, for the methods involved in
the usage automaton. E.g. in the usage automaton plugin-out above, the event read is fired whenever a new object of the class FileInputstream is created, or a cookie is accessed through the method readCookie. Similarly, the event write is fired when the method getOutputStream is invoked on a Socket. The remaining fields describe the logics of the automaton. The tag states is for the set of states, start is for the initial state, and final is for the final state, denoting a policy violation. The tag trans precludes the transition relation of the automaton. In our example, a transition from q1 to q2 occurs upon opening any file or cookie. A transition from q2 to fail occurs upon opening an output stream on a socket. Since fail is offending, this indeed implements the first confinement policy.

The second policy is specified by the usage automaton plugin-cookie, above on the right, which introduces further peculiar features of Jalapa: parameters and guards. We start from the state qo. The event init(p, u), signalling the creation of a new plugin p with codebase URL u, causes a transition to q1. Upon a start(p), i.e. when p is launched by the browser, we reach q2. There, all the accesses to a cookie having a URL different from u lead to the offending state. When the control is transferred to another plugin, we reset the state to q1. At run-time, the policy plugin-cookie is enforced for all the possible instantiations of the formal parameters p, u, and u'. Since this policy spans over multiple activations of plugins, we enforce it globally throughout the execution of the browser.

Once the needed policies and sandboxes have been defined, the next step is to instrument the compiled program with the hooks from the security-relevant methods to the execution monitor. Our tool implements this step as a bytecode transformation, discussed in more detail below. The resulting bytecode will respect all the usage policies at hand, within their scopes (see [10] for usage details). In [2] we formally prove that the run-time mechanism implemented by Jalapa is sound and complete w.r.t. the specification of policy compliance.

The Jalapa bytecode instrumentator. Our approach to code instrumentation is based on class wrapping, at the bytecode level. Since this solution suffers from some known issues, when moving to a production implementation we plan to follow a bytecode rewriting approach à la Kava [12]. First, we detect the set M of all the methods involved in policies. We inspect the bytecode, starting from the methods used in the aliases, and then computing a transitive closure through the inheritance graph. We create a wrapper for each of these methods. A wrapper WC for the class C declares exactly the same methods of C, implements all the interfaces of C, and extends the superclass of C. Indeed, WC can replace C in any context, in that it admits the same operations of C. A method m of WC can be either monitored or not. If the corresponding method m of C does not belong to M, then WC simply calls C.m. Otherwise, WC calls the PolicyPool.check method that controls whether C.m can actually be executed without violating the active policies. A further step substitutes (the references to) the newly created classes for (the references to) the original classes. Finally, the instrumented code is linked to the Jalapa run-time support, implementing the active policy in the browser. Since this policy spans over multiple activations of plugins, we enforce it globally throughout the execution of the browser.

As the workshop chairs, we would like to thank the program committee, whose invaluable help and enthusiasm ensured the success of the event. We would also like to thank all anonymous referees, for their hard work, particularly as much of this had to be done over their Christmas holidays.

March 2009

Elvira Albert
Samir Genaim
We illustrate our methodology for securing Java programs, as well as some key features of Jalapa, with the help of an example. Suppose you have a simple Web browser whose functionality can be extended with plugins, and with methods for handling connections and cookies. Since plugins can be downloaded from the network, possibly from untrusted sites, we want to control their behaviour, and block their execution at the moment they attempt some malicious action. In particular, we focus here on two confinement policies, that prevent plugins from transmitting data read from the local file system, either directly or by exploiting cookies to implement a covert communication channel (although stronger, these policies imply non-interference). Before formally specifying these policies, we consider a skeletal implementation of the classes Browser and Plugin.

```java
public class Browser {
    private Map<URL, String> cookies;
    public Browser() {
        cookies = new HashMap<URL, String>();
    }
    public void connect(URL url) throws Exception {
        URLConnection uc = url.openConnection();
        out = new BufferedWriter(new OutputStreamWriter(uc.getOutputStream()));
    }
    public void writeCookie(URL u, String msg) {
        cookies.put(u, msg);
    }
    public String readCookie(URL u) {
        return cookies.get(u);
    }
}

public abstract class Plugin implements Runnable {
    Plugin(Browser browser, String name, URL codebase) { ... }
    public void doIt() {
        try {
            // invokes this.run() within the sandbox
            PolicyPool.sandbox("plugin-out", this);
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

We assume that browser plugins extend the Plugin abstract class, by implementing the method doIt(). The browser starts a plugin by invoking the method doIt(), which is quite peculiar. Actually, it defines a sandbox, which will enforce the policy plugin-out throughout the run of the plugin. This means that all the security-relevant methods called while executing the method run() will be monitored, and blocked if not conformant to the policy. This policy is specified by the usage automaton plugin-out below on the left, to be discussed in a while.

A usage automaton closely resembles a finite state automaton. The field tagged name just defines the name of the policy. The tag aliases defines a mapping from the signatures of security-relevant methods to events that trigger the transitions of
that guard access to critical resources. At run-time, an access authorization is granted when all the methods on the call stack have the required permission (a special case is that of privileged calls, that trust the methods below them in the call stack). Being strongly biased towards implementation, this mechanism suffers from some major shortcomings. For instance, since a method removed from the call stack no longer affects security, stack inspection does not offer any protection when trusted code uses objects supplied by untrusted code [8].

Although many security policies are not enforceable by stack inspection, at present Java offers no other facilities to specify and enforce user-defined policies. Therefore, it is common practice to renounce to separating duties between functionality and security, and to implement the needed enforcement mechanism with local checks explicitly inserted into the code by programmers. Since forgetting even a single check might compromise the security of the whole application, programmers have to inspect their code very carefully. This may be cumbersome even for small programs, and it may also lead to unnecessary checking.

History-based security has been repeatedly proposed as a replacement for stack inspection [3,7,11]. Clearly, the ability of checking the whole execution history, instead of the call stack only, places history-based mechanisms a step forward stack inspection, from the expressivity viewpoint. However, since many possible history-based models can be devised, it is crucial to choose one which wisely conciliates the expressive power with the theoretical properties enjoyed. It is also important that the security mechanism can be implemented in a way that makes it transparent to programmers, and with a negligible run-time overhead.

Jalapa advocates local usage policies [3] as a history-based model for securing Java applications. Some remarkable features of Jalapa are that:

- local usage policies are expressive enough to model security requirements of real-world applications. For instance, we used them to specify the typical set of policies of a realistic bulletin board system [10].
- at the same time, usage policies are simple enough to be statically amenable, e.g. they can be model-checked against abstractions of program usages [4].
- local usage policies generalise global policies and local checks. The ability of sandboxing a piece of code by localizing the scope of a policy is particularly relevant, as the current programming methodologies provide for reusing code, and for exploiting services and components, offered by untrusted third parties.
- apart from the localization of sandboxes, enforcing policies is completely transparent to programmers.
- since the enforcement mechanism is based on bytecode rewriting, it does not require a custom Java Virtual Machine.
- even when the program source code is unavailable, Jalapa allows for specifying and enforcing policies on its behavior, by directly modifying the bytecode.

This paper gives an overview of Jalapa. We start by presenting our methodology for securing Java applications through local usage policies, with the help of some examples. Then, we give some insights about the design and the implementation of our tool, and we summarise the artifacts supporting our tool. We conclude by highlighting some of the present and future challenges of Jalapa.
Jalapa: Securing Java with Local Policies

Massimo Bartoletti
Department of Mathematics and Informatics, University of Cagliari, Italy

Gabriele Costa
Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Italy

Roberto Zunino
Department of Engineering and Information Science, University of Trento, Italy

Abstract
We present Jalapa, a tool for securing Java bytecode programs with history-based usage policies. Policies are defined by usage automata, that recognize the forbidden execution histories. Usage automata are expressive enough to allow programmers specify of many real-world usage policies; yet, they are simple enough to permit formal reasoning. Programmers can sandbox untrusted pieces of code with usage policies. The Jalapa tool rewrites the Java bytecode by adding the hooks for the mechanism that enforces the given policies at run-time.

Keywords: Usage control, history-based security, bytecode rewriting

1 Introduction
Security has been a major concern in the design and implementation of Java, starting from its early incarnations. Building upon the “safety pillars” of bytecode verification and secure class loading, new defence mechanisms have been developed over the years.

With the release of the JDK 1.0, a mechanism was provided to run untrusted mobile code into a sandbox with limited computational functionalities. The default sandbox prevented untrusted code from, e.g. accessing the local file system, from redefining the security manager (otherwise one could circumvent the sandbox), from connecting to (or accepting a connection from) any URL other than the one the code was downloaded, etc. Although these functionalities were completely customizable, this required to subclass the security manager, making it difficult to separate the functional aspects of programming from the security aspects.

While retaining the basic sandbox model of the JDK 1.0, the JDK 1.1 featured a “black or white” security model, based on digital signatures. Java-enabled browsers could be configured to trust digitally-signed mobile code, provided that the signature was put by a trusted entity. Trusted code were granted full privileges, while untrusted code were run without any privilege.

Starting with the JDK 1.2, a more fine-grained mechanism was devised, based on stack inspection [9]. This provides for associating methods with “protection domains” that reflect their provenance, and for defining a global security policy that grants each protection domain a set of permissions. Code includes local checks.
External Reviewers

Crégut, Pierre
Gotlieb, Arnaud
Haack, Christian
Hubert, Laurent
Hurlin, Clément
Jensen, Thomas
Loitsch, Florian
Navar, Jorge
Petri, Gustavo
Ruttenier, Philipp
Russo, Alejandro


termination analysis is not important and the running time of BinTerm is smaller on $P_{CLP}'$ than on $P_{CLP}$. For JLex, BinTerm is more than twice faster on $P_{CLP}'$ than on $P_{CLP}$, as the non-termination analysis removes 204 clauses from $P_{CLP}$ out of 356; among the removed clauses, there is a huge SCC containing 122 clauses where the arity of the involved predicate symbols is 8, which explains the gain in efficiency. On the contrary, the clauses removed for Kitten are several but include relatively small components and have small arity, so that the gain in efficiency is not significant there. This is because the cost of the termination analysis increases significantly with the arity of the predicates and, by removing clauses with small arity, we do not affect very much the efficiency of the termination analysis.

6 Conclusion

In this paper, we have presented some experiments with the automatic derivation of non-termination proofs for Java bytecode programs. When the approximation of the bytecodes into a path-length constraint is exact, the non-termination of the original program can be deduced from that of its CLP translation. When the approximation is not exact, it may happen that the bytecode program terminates while its CLP version does not terminate (Section 4.4 illustrates this situation). As a future work, we plan to replace some non-exact approximations (such as that of the getfield instruction or of the non-linear arithmetic operations) with exact ones that are suitable for deriving non-termination proofs of Java bytecode programs. To that purpose, a possibility is that of finding specific executions that make the program diverge, instead of proving a universal non-termination. In that direction, we might make some program variables ground, hence linearising some operations. This would be similar to the technique used in [6].

We have also implemented the non-termination tests of Section 3 in order to help the termination prover BinTerm used in the tool Julia+BinTerm. The results we have presented in Section 5 are encouraging; even for some large Java bytecode programs, the computational overhead of the non-termination analysis is unimportant; moreover, the termination prover BinTerm runs much faster when the components detected as non-terminating are removed from the CLP version of the original bytecode program.

References


Contents

Session 1

From Stack Maps to Software Certificates (invited talk) . . . . . . . . . . . 1
Thomas Jensen

Pervasive Low-Level Transformation for Transparently Distributed Java . . 3
Phil McGachey, Antony L. Hosking, J. Eliot B. Moss

Session 2

Virtual-Machine Abstraction and Optimization Techniques . . . . . . . . . 19
Stefan Brauchharter
Towards an XML-Based Byte Code Level Transformation Framework . . . 31
Arno Puder, Jessica Lee

The S3MS.NET Run Time Monitor . . . . . . . . . . . . . . . . . . . . . . 47
Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, Dries Vanoverberghe

Session 3

Soundly Handling Static Fields: Issues, Semantics and Analysis . . . . . . 53
Laurent Hubert, David Pichardie

User-Definable Resource Usage Bounds Analysis for Java Bytecode . . . . 69
Jorge Navas, Mario Mendez, Manuel V. Hermenegildo

An Ahead-of-Time Yet Context-Sensitive Points-to Analysis for Java . . . 87
Xin Li, Mizuhito Ogawa

Session 4

Using CLP Simplifications to Improve Java Bytecode Termination Analysis 103
Fausto Spoto, Lunjin Lu, Fred Mesnard

The Non-Interference Protection in BML . . . . . . . . . . . . . . . . . . . 119
Aleksy Schubert, Daria Walkiewicz-Chrupaæ

Experiments with Non-Termination Analysis for Java Bytecode . . . . . 135
Etienne Payet, Fausto Spoto

Jaklap: Securing Java with Local Policies . . . . . . . . . . . . . . . . . . . 149
Massimo Bartoletti, Gabriele Costa, Roberto Zunino
5 Using non-termination proofs to support termination analysis of Java bytecode

A completely different use of our non-termination tests consists in proving the non-termination of clauses of the $P_{CLP}$ program generated during the termination analysis of a Java bytecode program $P$. By removing such clauses, which cannot have any termination proof, we help the termination checker by simplifying its task. Since our non-termination tests are extremely efficient, while a thorough quest for a termination proof is in general expensive, the trade-off is positive and we get a more efficient termination analysis still keeping the same precision.

In particular, we have implemented the non-termination tests of Section 3 to help the termination prover $BinTerm$ used in the tool $Julia+BinTerm$ [19]. Given a Java bytecode program $P$, our approach consists in a preliminary analysis which considers the strongly connected components (SCCs) of $P_{CLP}$; any SCC where a non-terminating ground query is found is removed from $P_{CLP}$ and the resulting $CLP$ program $P'_{CLP}$ is analysed by $BinTerm$.

We have run $Julia+BinTerm$ on the following Java bytecode programs using a Linux machine based on a 2.33GHz Intel Core 2 Duo with 2 gigabytes of RAM.

<table>
<thead>
<tr>
<th>$P$</th>
<th>number of methods in $P$</th>
<th>number of clauses in $P_{CLP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCup</td>
<td>270</td>
<td>170</td>
</tr>
<tr>
<td>JLex</td>
<td>137</td>
<td>356</td>
</tr>
<tr>
<td>Kitten</td>
<td>2149</td>
<td>1224</td>
</tr>
</tbody>
</table>

The next table summarizes the results. For each program $P$, it reports: the number of clauses removed from $P_{CLP}$ by the non-termination analysis; the non-termination analysis time; the $BinTerm$ running time on $P'_{CLP}$; the $BinTerm$ running time on $P_{CLP}$. All the times are in seconds.

<table>
<thead>
<tr>
<th>$P$</th>
<th>clauses removed</th>
<th>non-termination analysis</th>
<th>$BinTerm$ on $P_{CLP}$</th>
<th>$BinTerm$ on $P'_{CLP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCup</td>
<td>113</td>
<td>0.09s</td>
<td>3.90s</td>
<td>5.66s</td>
</tr>
<tr>
<td>JLex</td>
<td>204</td>
<td>0.20s</td>
<td>21.20s</td>
<td>55.30s</td>
</tr>
<tr>
<td>Kitten</td>
<td>288</td>
<td>0.68s</td>
<td>99.52s</td>
<td>100.99s</td>
</tr>
</tbody>
</table>

In these experiments on large programs, the computational overhead of the non-
(true denotes the always satisfiable constraint) does not terminate, hence we safely conclude that the program List does not terminate either.

4.4 Non-exact approximations

Consider the mul bytecode instruction that removes the two top operand stack elements and replaces them with the result of their multiplication. As there is no linear way of expressing a constraint on the result of the multiplication, we just set

\[ \text{mul}_{IL2}^p = \text{Unchanged}, \#l, \#s - 2 \]

(\#l and \#s are the number of local variables and stack elements at program point \( q \) where the instruction occurs) meaning that the instruction does not modify any local variables nor any stack element which are not its operands; however, no constraint on the new top of the stack (the result of the multiplication) is generated. The Java program:

```java
public class Field {
    private int n = 6;
    public static void main(String args[]) {
        Field f = new Field();
        for(int i = 2; i < f.n; i++);
    }
}
```

terminates. Notice that the guard of the loop involves a multiplication. The corresponding CLP(IL2) program \( \text{Mul}_{CLP} \)

\[ \text{entry}(IL2) = \{ \text{IL2} = \text{OL2} = 1, \text{IL2} \leq 2 \}, \text{entry}(OL2) \]

does not terminate. Indeed, the projection of the constraint of the unique clause of \( \text{Mul}_{CLP} \) onto \( \text{IL2} \) (resp. \( \text{OL2} \)) is \( \text{IL2} \geq 2 \) (resp. \( \text{OL2} \geq 3 \)) and we have

\[ \text{Set}((\text{entry}(OL2)) \cup \text{OL2} \geq 3) \subseteq \text{Set}((\text{entry}(IL2)) \cup \text{IL2} \geq 2) \].

Therefore, by Theorem 3.1, the non-empty set \( \text{Set}((\text{entry}(IL2)) \cup \text{IL2} \geq 2) \) is non-terminating \( \text{Mul}_{CLP} \). However, the non-termination of \( \text{Mul} \) does not follow from this result, since we are using approximated constraints.

We are facing a similar situation when dealing with numeric fields. The `getfield` instruction takes the reference to an object \( o \) located on top of the stack and replaces it with the value of \( o.f \). In [55] we defined

\[ \text{getfield}^p_{IL2} f = \text{Unchanged}, \#l, \#s - 1 \]

wherever the field \( f \) has integer type (\#l and \#s are the number of local variables and stack elements at program point \( q \) where the instruction occurs). No constraint is generated for the new top of the operand stack (the value of the field) since its path-length is unknown. The Java program:

```java
public class Field {
    private int n = 6;
    public static void main(String args[]) {
        Field f = new Field();
        for(int i = 2; i < f.n; i++);
    }
}
```

From Stack Maps to Software Certificates

Thomas Jensen

CNRS
IRISA, Campus de Beaulieu, F-35042 Rennes, France

One of the most successful applications of the principle of Proof Carrying Code has been to simplify the byte code verification of Java byte code. By providing additional typing information with a class file (in the form of the so-called stack maps), the task of byte code verifier has been simplified. Initially meant to facilitate byte code verification on embedded, resource-constrained devices, the idea has now been adopted in standard Java.

In this talk, we’ll review the basic principles behind Lightweight Byte Code Verification and the notion of stack map. We’ll then demonstrate how the idea of sending a certificate along with the byte code to facilitate verification can be extended to other kinds of security-related properties, explain how the information can be compressed into suitable stack maps, and discuss to what extent such an extended byte code verifier can be incorporated into a Trusted Computing Base through certification inside a proof assistant.

References


This paper is electronically published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
The CLP(PL) program \(\text{Rec1}\) CLP:
\[
\text{entry}(IL_0) \leftarrow \{ IL_0 - OL_0 = 1, \ IL_0 \geq 1, -IL_0 \geq -2 \}, \text{entry}(OL_0)
\]
terminates, hence by Theorem 2.1 we conclude that \(\text{Rec1}\) terminates. If we turn \(\text{Rec1}\) into the following non-terminating program (where the programmer forgot the base case in the recursive method):

```java
public class Rec2 {
    public static int sum(int n) {
        return n + sum(n-1);
    }
    public static void main(String args[]) {
        sum(2);
    }
}
```

we get the CLP(PL) program \(\text{Rec2}\) CLP:
\[
\text{entry}(IL_0) \leftarrow \{ IL_0 - OL_0 = 1, -IL_0 \geq -2 \}, \text{entry}(OL_0)
\]
By Theorem 3.1, \(\text{Rec2}\) CLP does not terminate. As the approximation of the bytecode program corresponding to \(\text{Rec2}\) is exact, we can safely conclude that \(\text{Rec2}\) does not terminate either.

### 4.3 Exact approximations with data structures

All examples above deal with integer values only. Let us consider the following program now, where a list data structure is defined and recursively scanned:

```java
public class List {
    private int head;
    private List tail;
    public List(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
    private void iter() {
        if (tail != null) iter();
    }
    public static void main(String args[]) {
        List l = new List(0, new List(1, null));
        l.iter();
    }
}
```

The method \texttt{iter} (intended to perform an iteration over a list) contains a bug since it recurs on the same list rather than on its tail (\texttt{iter()} instead of \texttt{tail.iter()}).

The bytecode version of this program has an exact approximation as our cyclicity analysis correctly infers that the list \(l\) in the method \texttt{main} is not cyclical. The corresponding CLP(PL) program \(\text{List}\) CLP:
\[
\text{entry} \leftarrow \text{true}, \text{entry}
\]
the path-length analysis and no clause is generated with a false constraint. Indeed, such clause would not influence the termination or non-termination behaviour of the program, since it would just stop the CLP resolution process. Indeed, by applying Theorem 3.1 to the recursive clause we get that Set(\(Q\)) is non-terminating w.r.t. Add3\_CLP, where

\[
Q := \langle \text{block}(IL3) \mid IL3 \geq -2 \rangle.
\]

Notice that we have to infer a non-terminating query of the form \(\langle \text{entry}(-) \rangle \cdots\) to conclude the non-termination of Add3\_CLP because the entry point of the loops of the program is the predicate entry. The projection of the constraint of the first clause onto OL3 is OL3 = 2 and we have

\[
\text{Set}(\langle \text{block}(OL3) \mid OL3 = 2 \rangle) \subseteq \text{Set}(Q).
\]

Hence, by Theorem 3.2 applied to the first clause of Add3\_CLP and to Q, we have that Set(entry(IL3) | true) is non-terminating w.r.t. Add3\_CLP, (where true denotes the always satisfiable constraint). Therefore, Add3\_CLP does not terminate so we conclude that Add3 does not terminate either.

If we nest the non-terminating loop of program Add3 into a separated method, such as in:

```java
public class Add3 {
    public static void loop(int k) {
        for(int i = 2; i < 2 + k; i--);
        public static void main(String args[]) {
            loop(3);
        }
    }
}
```

we get the CLP(IL) program Add4\_CLP:

\[
\text{entry}(IL1) = \{ IL1 - OL1 = 1, -IL1 \geq -2 \}, \text{entry}(OL1)
\]

which does not terminate (by Theorem 3.1). Hence we conclude that Add4 does not terminate either.

4.2 Exact approximations with recursion

The following terminating Java program involves a recursive method:

```java
public class Add4 {
    public static int sum(int n) {
        if (n <= 0) return 0;
        else return n + sum(n-1);
    }
    public static void main(String args[]) {
        sum(2);
    }
}
```

**Abstract**

The transformation of large, off-the-shelf Java applications to support complex new functionality essentially requires generating an entirely new application that retains the execution semantics of the original. We describe such a whole-program modification in the context of RuggedJ, a dynamic transparent Java distribution system.

We discuss the proxy-based object model that allows remote Java objects to be referenced in the same way as those residing on the current virtual machine, the optimizations that allow us to bypass proxies in the case of purely local or remote object, and the mechanisms needed to guarantee that static data remain unique in a distributed system. We then detail some of the more interesting features involved when implementing this object model in rewritten bytecode, including transformations required within method bodies and coordination between bytecode and the run-time system that distributes an application across the network.

**Keywords:** Java, Bytecode Transformation, Load-Time Rewriting, Transparent Distribution, Object Model

1 Introduction

Automatic code modification for Java applications is a widely-used technique that adds functionality to existing software. Aspect-oriented programming or bytecode rewriting make it trivial for programmers to implement cross-cutting concerns such as logging, error handling, or profiling without modification to original applications. More complex is the comprehensive transformation of an application; generating an entirely new program that retains the execution semantics of the original, while adding substantive new functionality.

In this paper we describe the process of pervasive transformation in our transparently distributed Java system, RuggedJ. We use load-time dynamic bytecode transformation to generate an entirely new class hierarchy that mirrors the structure of an off-the-shelf Java application, adding the necessary functionality to execute the application across a network of Java virtual machines. While we discuss our program transformation process in the context of RuggedJ, many of our techniques would be equally useful to other large-scale modification applications such as persistence.

We discuss a proxy-based object model that abstracts object implementation, hiding whether they are local or remote to a given virtual machine. This model allows for objects to be distributed and migrated across the RuggedJ network while still preserving the execution semantics of the original. We describe such a whole-program transformation process in the context of RuggedJ, a dynamic transparent Java distribution system.

This paper is electronically published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs
semantics and class hierarchy of the original application. Additionally, by referring to transformed classes in rewritten bytecode only using interfaces we allow the elimination of proxies in the common case where an object is known to be always local or remote.

Performing our transformations at the bytecode level affords us several major advantages: We do not require access to the original application source; we can perform our transformation at class-load time, taking advantage of dynamic knowledge of the execution environment; the relative simplicity of bytecode when compared to source allowing our rewrites to be more general; and we can perform incremental changes on-the-fly, referring to generated classes that may or may not have been created on demand without having to perform a whole-program compilation.

## 2 Application Distribution with RuggedJ

The current trend in microprocessor technology is for increases in the number of cores on a processor to replace the once-familiar increases in processor speed. The immediate implication for application developers is that we can no longer rely on the next generation of processors to make our systems run faster; we must instead take advantage of parallelism on individual machines and, increasingly, distribution across clusters of commodity machines.

Unfortunately, the implementation of complex distributed systems demands a great deal of additional effort from programmers and is liable to introduce obscure bugs. Objects must be allocated and tracked across nodes, method calls and field accesses must take into account the location of their targets, objects may need to be migrated from node to node in order to gain acceptable performance, and so forth.

RuggedJ is an automatic transparent distribution system that aims to eliminate these concerns by transforming a Java application to run across a cluster of machines. We achieve this through a combination of a run-time distribution library and a transformation process that creates a new, distribution-aware, application from a set of standard Java class files.

Our implementation of RuggedJ is mostly complete. The bytecode transformation process is in place and tested on realistic applications running on a single node. We have distributed simple applications, but are currently working on the complete distribution of complex systems.

### 2.1 The RuggedJ Network

A RuggedJ network consists of a set of Java virtual machines (VMs) that distribute and run an off-the-shelf Java application. Each virtual machine (a node, in RuggedJ terminology) contains an instance of the RuggedJ run-time library that interacts with the run-times on other nodes to coordinate the execution of an application. Figure 1 shows the construction of a RuggedJ network:

We refer to a single physical machine available to RuggedJ as a host. This is distinct from a node; a single host can run multiple nodes. RuggedJ is designed to be platform-agnostic, in that a network can consist of heterogeneous hosts. We require only that each host is capable of running a fully-functional Java VM, and that all VMs run the same version of Java, including the standard class libraries.

Each node consists of two parts: the transforming class loader and the run-time library. The presence of a transforming class loader on each node allows a given class to bytecode and the approximation above, we must have that the local variables have the same values in $\hat{\sigma}$ and $\hat{\sigma}$ and the top of the stack of $\hat{\sigma}$ is the sum of the topmost two values on top of the stack of $\sigma$, so that these states are such that $\text{add}_4(\sigma) = \sigma$.

The corresponding CLP(FL) program $Add3_{CLP}$ is:

```plaintext
entry(IL3) ← (IL3 = 0), block(IL3)
block(IL3) = (IL3 = 0)
```

The predicate `entry` denotes the entry point of the loop; the local variable $i$ implements its value while variable $k$ has been removed since it is irrelevant for the termination of the program. This CLP program has been derived by using the abstract interpretations cited in the introduction. Namely, we have used the path-length abstract analysis, which has derived the constraint $IL2 = OL2 = 0$ (that is, local variable $2$, which is $1$, decreases along iterations of the loop) and the constraints $IL2 \geq -4$, $IL2 \geq 2$, which provide bounds on the possible values of that variable inside the loop. That CLP program terminates. By Theorem 2.1 we conclude that $Add2$ terminates also. If we turn $Add1$ into the non-terminating program:

```plaintext
public class Add2 {
    public static void main(String args[]) {
        int k = 3;
        for(int i = 2; i < 2 + k; i--);
    }
}
```

we get the CLP program $Add2_{CLP}$:

```plaintext
entry(IL2) ← (IL2 = OL2 = 0, IL2 = 0), entry(OL2)
```

which by Theorem 3.1 does not terminate because the projection of the constraint of its unique clause onto $IL2$ (resp. $OL2$) is $IL2 \geq 2$ (resp. $OL2 \geq -1$) and we have:

```plaintext
Set(entry(OL2)) − OL2 = 1 \subseteq Set(entry(IL2)) − IL2 = -2
```

Here, we can safely conclude the non-termination of $Add2$ from that of $Add2_{CLP}$.

Our technique is also able to handle more complicated situations. For instance, if we nest the non-terminating loop of program $Add2$ into a terminating loop, we get:

```plaintext
public class Add3 {
    public static void main(String args[]) {
        int k = 3;
        for(int j = 0; j < 10; j++)
            for(int i = 2; i < 2 + k; i--);
    }
}
```

The corresponding CLP(FL) program $Add3_{CLP}$:

```plaintext
entry(IL3) ← (OL3 = 2), block(IL3)
block(IL3) = (IL3 = 0)
```

does not terminate. Note that the outer loop does not appear in the CLP program, since the exit condition $i \geq 2 + k$ of the inner loop is found to be false during...
recursive clauses of the program and then complete this set with the help of Theorem 3.2.

4 Proving non-termination of Java bytecode programs

In this section, we give several examples of situations where we can conclude the non-termination of the original program from that of the CLP program, as well as examples where instead this is not possible.

4.1 Exact approximations with iterations

When the approximation into a path-length constraint of the Java bytecode program \( P \) under analysis is exact, a proof of non-termination of \( P_{CLP} \) is also a proof of non-termination of \( P \). The formal definition of exact requires the bytecodes to have a concrete behaviour which is exactly matched by their numerical abstraction, that is, every pair of states satisfying the input/output abstraction of the bytecode must correspond to an actual, concrete behaviour of the bytecode. Note that the converse must always hold by the correctness of the abstraction.

Definition 4.1 [Exact Abstraction] Let \( ins \) be a bytecode instruction, formalised as an input/output map on concrete JVM states, as in [19], and let \( ins^{\mathcal{D}} \) be a correct approximation of its behaviour, i.e., a constraint over input variables \( \mathcal{s} \) and output variables \( \mathcal{v} \). This approximation is exact if and only if, for all input states \( \mathcal{s} \) and output variable \( \mathcal{v} \) satisfying the static information at \( ins \) (number and type of local variables and stack elements), whenever \( \mathcal{v} = \text{pathlength}(\mathcal{s}(\mathcal{v})) \), then \( \mathcal{v} = \mathcal{v} \).

Consider for instance the program \( Add1 \):

```java
public class Add1 {
    public static void main(String args[]) {
        int k = 3;
        for(int i = 2; i < 2 + k; i++);
    }
}
```

The approximation of the bytecode program corresponding to \( Add1 \) is exact: the loop guard involves the \( add \) bytecode instruction whose approximation, as provided in [19], is

\[
\text{add}^{D} = \text{Unchanged}(\#l, \#s - 2) \cup \{ \#s - 2 \}
\]

where \( \#l \) and \( \#s \) are the number of local variables and stack elements at program point \( q \) where the instruction occurs; we distinguish between variables \( s \) at the beginning of the execution of the bytecode, written as \( s \), and variables at its end, written as \( \mathcal{v} \). The formula above means that \( add \) does not modify any local variable nor any stack element not involved in the addition; moreover, the new top of the stack \( \#s - 1 \) holds a value which is equal the addition of the former two topmost stack elements \( \#s - 2 \) and \( \#s - 1 \). This approximation is exact since, for every couple of input state \( \mathcal{s} \) and output state \( \mathcal{v} \) satisfying the static information at this

be rewritten differently on different nodes, taking advantage of the capabilities of the host or knowledge of the execution environment. We designate the node upon which the application starts to be the head node. As well as functioning as a central location for the application, handling I/O requests and other operations that require native access to a particular host. It also acts as an overseer for coordination between nodes: the head node maintains full information on the location of objects and the condition of the network, providing other nodes with a definitive source for this information. While the head node can present a bottleneck, it is necessary not only as a co-ordinator but also as a location for classes that cannot be distributed (see Section 3.4).

2.2 Application Partitioning

The partitioning strategy for a given application is defined by the application developer. While substantial research exists in the literature concerning automatic application partitioning [9, 13, 19], we feel that one is more likely to arrive at an optimal partitioning when developing it using the domain-specific application knowledge available to the programmer. Additionally, many automatic partitioning schemes rely on an advance knowledge of the network configuration under which an application will run. This runs counter to our aim of environmental flexibility, where one can use an application partitioning on a RuggedJ network made up of arbitrary nodes.

To allow the partitioning developer to take full advantage of the RuggedJ network available, we provide a plug-in interface to which one can attach a partitioning strategy, allowing access to the RuggedJ run-time’s dynamic internal state (network conditions, node load levels, etc.). RuggedJ consults the strategy for the running application both at class-loading time, guiding the rewrites that the system applies to a class, and at run time, allowing it to base its dynamic decisions upon the current state of the network. Some of the options available to the partitioning designer are discussed in Section 4.4.

2.3 Run-Time Support

The RuggedJ run-time library manages the interaction between rewritten bytecode on remote nodes, allowing separate processes to interact and execute a single application. Parts
We let \( \bar{t} \) denote a sequence of terms, \( \bar{x} \) and \( \bar{y} \) denote sequences of distinct variables, \( p \) and \( q \) denote predicate symbols and \( c \) denote a path-length constraint. An atom has the form \( p(\bar{x}) \) where the length of \( \bar{x} \) equals the arity of \( p \). A query has the form \( \langle p(\bar{x}) \rangle \). A clause has the form \( p(\bar{x}) \leftarrow c, q(\bar{y}) \) where \( \bar{x} \) and \( \bar{y} \) are disjoint and the variables occurring in \( c \) necessarily occur in \( \bar{x} \cup \bar{y} \). A CLP(PL) program is a finite set of clauses. We use \( \exists c \) as a shorthand for \( \exists X_1 \ldots \exists X_n \) where \( X_1, \ldots, X_n \leftarrow \bar{x} \). The projection of \( c \) onto the sequence \( \bar{x} \) is denoted by \( \exists \bar{x} c \) and is the constraint \( \exists \bar{x} c \leftarrow c \) where \( \bar{x} \) is the set of variables occurring in \( c \). The set described by a query \( Q \leftarrow \langle p(\bar{x}) \rangle \) is denoted by \( \text{Set}(Q) \); it consists of all the atoms of the form \( p(\bar{x}(X_1), \ldots, v(X_n)) \) where \( X_1, \ldots, X_n \leftarrow \bar{x} \) and \( v \) is a ground solution of \( c \). We say that \( \text{Set}(Q) \) is non-terminating wrt. a CLP(PL) program \( P \) when for all \( p(\bar{x}(X_1), \ldots, v(X_n)) \in \text{Set}(Q) \), the query
\[
\langle p(X_1, \ldots, X_n) \mid X_1 = v(X_1), \ldots, X_n = v(X_n) \rangle
\]
is non-terminating wrt. \( P \) by using the standard semantics of constraint logic programs. This means that an infinite computation can be built for that query in the program \( P \). Note that we do not consider any precedence between the clauses of \( Q \) that is, we assume a non-deterministic resolution of a predicate with all the clauses that define that predicate. The following results provide simple non-termination conditions for constraint logic programs.

**Theorem 3.1** (110) Let \( p(\bar{x}) \leftarrow c, p(\bar{y}) \) be a recursive clause in a CLP(PL) program \( P \). If \( \text{Set}(\langle p(\bar{y}) \rangle \leftarrow \exists \bar{y} c) \subseteq \text{Set}(\langle p(\bar{x}) \rangle \leftarrow \exists \bar{x} c) \) then \( \text{Set}(\langle p(\bar{x}) \rangle \leftarrow \exists \bar{x} c) \) is non-terminating wrt. \( P \). □

**Theorem 3.2** (110) Let \( q(\bar{x}) \leftarrow c, q(\bar{y}) \) be a clause in a CLP(PL) program \( P \) and \( Q \) be a query such that \( \text{Set}(Q) \) is non-terminating wrt. \( P \). If \( \text{Set}(\langle p(\bar{y}) \rangle \leftarrow \exists \bar{y} c) \subseteq \text{Set}(Q) \) then \( \text{Set}(\langle p(\bar{x}) \rangle \leftarrow \exists \bar{x} c) \) is non-terminating wrt. \( P \). □

The intuition of Theorem 3.2 is that any value \( q(\bar{x}) \) in \( \text{Set}(\langle p(\bar{y}) \rangle \leftarrow \exists \bar{y} c) \) satisfies \( \exists \bar{x} c \), the guard of the clause, and the corresponding output \( p(\bar{x}) \) is included in \( \text{Set}(\langle p(\bar{y}) \rangle \leftarrow \exists \bar{y} c) \). As \( \text{Set}(\langle p(\bar{y}) \rangle \leftarrow \exists \bar{y} c) \subseteq \text{Set}(Q) \) and \( \text{Set}(Q) \) is non-terminating wrt. \( P \), then \( p(\bar{x}) \) does not terminate wrt. \( P \), so \( q(\bar{x}) \) does not terminate also.

These theorems provide a simple mechanism to infer ground non-terminating queries: first, use Theorem 3.1 to infer a set of non-terminating queries from the
translation of $P$ into $P_{CLP}$: both sharing and cyclicity analyses are approximated, so that, for instance, the analyser might not necessarily prove that a non-cyclical list is actually non-cyclical. Moreover, some bytecodes have an inherently non-linear behaviour, such as multiplications and divisions, and cannot hence be approximated by using the linear constraints available for the path-length.

The translation from Java or Java bytecode to $CLP$ makes it uniform the treatment of any kind of loops: for, while loops, recursion, loops having exit conditions depending on numerical, reference or Boolean variables, loops exiting become of the break statement, all become a loop in the graph of blocks of $P_{CLP}$. The termination of $P_{CLP}$ can hence be established in a uniform way, also in the presence of Boolean variable assigned inside an if statement and hence making a loop exit.

An important point about the program $P_{CLP}$ is that its termination is meaningful for ground inputs only, where all variables have been bound to their integer path-length (Theorem 2.1). Moreover, the clauses of $P_{CLP}$ are binary, that is, they have the form $p(X) \leftarrow c, q(Y)$, with only one predicate on the right.

The termination of $P_{CLP}$ is proved by the BinTerm tool by F. Mesnard, that finds decreasing measures across iterations of most loops in $P_{CLP}$. The computational cost of the tool decreases by reducing the number of clauses in $P_{CLP}$; namely, only clauses in a loop are considered, since they correspond to loops or recursion in the original program $P$ and are those that determine the termination or non-termination of the program. Moreover, its cost is reduced also by decreasing the arity of the predicates, when it is clear that the removed arguments are irrelevant for the termination of the predicates. These optimizations are defined and proved correct in [18]. As a consequence, in all our examples, the $CLP$ program will express the path-length relationships for the loops of the program only.

Although the converse of Theorem 2.1 does not hold in general, there are many cases when the approximation of the original program $P$ into path-length is exact, by which we mean that all denotations represented by the $P_{CLP}$ program are actual denotations that represent real, concrete executions of $P$. This is the case, for instance, of the approximations of the instructions dealing with integer values, with the notable exception of multiplications and divisions; as well as of instructions dealing with data structures that have been successfully proved to be non-cyclical by the cyclicity analysis. In those frequent cases, a proof of non-termination for the $CLP$ program induces a proof of non-termination for the original Java bytecode program.

In the following, we discuss how proofs of non-termination for $CLP$ programs can be constructed and exemplify many cases when we can conclude (or not) that the original Java bytecode program does not terminate either.

3 Proving non-termination of constraint logic programs

A non-termination criterion is provided in [10] for the standard operational semantics of constraint logic programming, where free variables may occur in a call to a predicate. The specialisation of this criterion to the semantics we consider in this paper (free variables are not allowed in a call to a predicate) is briefly described in this section.

We consider constraint logic programs over path-length polyhedra ($CLP(PL)$), class, the Java type system will recognize an object that implements this interface as the original class. This property simplifies the rewriting of certain bytecode structures, such as instanceof checks and exception handling, and removes the need to transform every reference to the original class to the rewritten class. The local, stub and proxy classes each implement the interface, and rewritten code refers to a class primarily via its rewritten interface.

**Class $X_{local}$**: The local class contains the fields and the implementation of each instance method from the original class. One can thus think of it as the “actual” object. Any methods invoked upon the object must ultimately execute on an instance of the local class.

**Class $X_{stub}$**: The stub class represents a remote object (i.e., one for which a local version exists on a different node). The stub contains a globally-unique identifier of the remote object, and it implements each method of the interface as a remote method invocation.

**Class $X_{proxy}$**: The proxy class provides a level of indirection between calling code and the local or stub implementation of an object. It contains a single field that holds a reference to either a local or stub object, and implementations of every method in the interface that invoke the relevant method on the referenced object.

Where the original application allocates an object of type $X$, the transformed version creates a pair of objects. One is either an $X_{local}$ or $X_{stub}$, depending on the node upon which the allocation occurs. The other is an $X_{proxy}$ object that references the local or stub object. By referring to proxies rather than local or stub object in rewritten code, RuggedJ ensures that only a single pointer exists to a local or stub object on a given node. This allows objects to migrate easily from node to node: should an object move from the local node, it is necessary only to update the reference in the proxy from the local class to a stub. Migration without proxies would require updating all references in objects or on stacks, which would be prohibitively expensive.

The design of the object model, however, does allow for the direct allocation of local or stub objects, bypassing the proxy. This is desirable for objects that are known never to migrate, such as those objects directly tied to the local virtual machine (such as file handles, class objects, and so forth), or objects known by the author of the partitioning strategy to exist on only one machine (such as temporary objects or local data structures). Allocating proxies for such objects would be unnecessary, adding the overhead of indirection when the referenced object is never going to change. In these cases, RuggedJ instead simply allocates either the local or stub object.

We can use $X_{proxy}$, $X_{local}$, and $X_{stub}$ objects interchangeably in this manner because each implements the generated interface $X$. We make all method calls within rewritten code in terms of the interface, and field accesses go through the generated get and set methods. By calling methods through interfaces, we minimize the transformation necessary on calling code, while maximizing flexibility in the types of objects used.

3.2 Inheritance

As well as providing a mechanism by which we can reference different versions of a class uniformly, RuggedJ’s generated interfaces maintain the inheritance relationships between original classes. Figure 3 shows the relationship between transformed classes (omitting static parts).

The original application’s inheritance relationship between subclass $Y$ of class $X$ appears as the transformed interface $Y$ extending interface $X$. Since rewritten code refers to
objects exclusively by interface, this allows one to use any object that implements \( Y \) when the original code required an instance of \( Y \). Similarly, checkcast or instanceof operations operate over interfaces, and produce the same results in transformed code as in the original application.

Each transformed class \( Y_{\text{local}}, Y_{\text{stub}} \) and \( Y_{\text{proxy}} \) extends the equivalent part of class \( X \). This is not necessary to preserve the inheritance relationships of the original application. Other than when allocating objects, rewritten code never refers to these individual classes. Rather, this subclassing works to simplify the implementation of these classes. Without it, each class would have to contain the fields and implementations for every method of the superclasses of its unmodified version, which would lead to duplication of code and overly-complex classes.

### 3.3 Static Classes

Turning to the static parts in Figure 2, we see that RuggedJ generates an additional interface and two classes:

**Interface \( X_{\text{static}} \):** This interface contains each static method and get/set methods for each static field in the original class. It functions similarly to the instance interface \( X \). Both \( X_{\text{static}} \), \( X_{\text{static}_\text{local}} \) and \( X_{\text{static}_\text{stub}} \) classes implement \( X_{\text{static}} \), allowing rewritten code to use them interchangeably.

**Class \( X_{\text{static}_\text{local}} \):** This class contains the static fields and implementations of each static method from the original class. RuggedJ modifies both fields and methods to be instance members of \( X_{\text{static}_\text{local}} \) rather than static members. This allows class \( X_{\text{static}_\text{local}} \) to fulfill the requirements of interface \( X_{\text{static}} \), and decouples the implementation of static members from the implementation of the VM.

**Class \( X_{\text{static}_\text{stub}} \):** The static stub acts similarly to the instance stub class. It contains implementations of each method in interface \( X_{\text{static}} \) that perform remote invocations on the appropriate \( X_{\text{static}_\text{local}} \) object.

Transforming static methods of original class \( X \) into instance members of class \( X_{\text{static}_\text{local}} \) serves two purposes. First, it allows the static part of an object to be treated as any other object in the RuggedJ network. This allows us to take advantage of any object migration or caching performed by the system for static data as well as individual objects. More importantly, however, is the fact that transforming static data to representation as an object allows us to ensure that only one copy of the data exists in the

---

**2 Compilation of Java bytecode into constraint logic programs**

Java bytecode is a low-level object-oriented type-safe language. Its static analysis is complicated by the fact that it has no explicit structure, differently from high-level languages, and that it uses a stack of temporary variables. Hence the number and type of the variables are different at different program points inside the same method.

We have recently developed a static analysis of Java bytecode programs (and hence of Java programs) that proves the termination of most methods of a program [19]. The idea is that the Java bytecode program is first translated into its basic blocks and then an abstract interpretation [8], based on a denotational semantics over those blocks, is applied by using different abstract domains of analysis. The latter provide a conservative approximation of the numerical and structural constraints on the numbers or data structure used by the program: a first domain, for sharing [13], determines when data structures bound to program variables might share locations on the heap, so that an update of one variable might also affect the others. This information is exploited in the second domain, for cyclicity [12], which determines when the data structure bound to a program variable might contain loops of locations, so that an iteration over that data structure might not necessarily terminate. Both kinds of information are then used in a path-length domain [17,19], that computes the relationship between the size of program variables before and after the execution of each instruction in the bytecode: the size or path-length of a variable bound to a data structure is the maximal length of pointers that one can follow from that variable; the path-length of a variable bound to an array is the length of the array; the path-length of a numerical variable is its value, the path-length of a Boolean variable is 0 for false and 1 for true. The result of the path-length is finally used to express the relationship between the size of the variables at the beginning and at the end of each basic block of the program. This is written in terms of a constraint logic program \( PCLP \) over linear constraints, whose predicates \( b(var) \) correspond to each basic block \( b \) of \( P \) and \( var \) are the variables at the beginning of the execution of \( b \). These approximations build constraints that are later used in order to derive bounds on the values of variables in programs, which is crucial for termination and non-termination analyses to work. The main result proved in [18], wrt. termination analysis, is the following:

**Theorem 2.1** Let \( P \) be a Java bytecode program and \( b \) a basic block of \( P \). If the query \( b(var) \) has only terminating computations in \( PCLP \), for all fixed integer values for \( var \), then all executions of a Java Virtual Machine started at \( b \) terminate.

The converse, however, does not hold in general: we can find programs \( P \) and a basic block \( b \) of \( P \) such that, in the translation \( P_{CLP} \), predicate \( b(var) \) does not terminate for some fixed initial integer values for \( var \), although all executions of \( P \) starting at \( b \) do terminate. This is due to the approximations done during the
As a consequence of the wide use of Java bytecode, research is increasingly focused on checking, in an automatic way, that Java bytecode applications are not harmful. This includes the proof that, for instance, they do not oversee the resources of the system. One such resource is time. In particular, proofs of termination of Java bytecode programs guarantee that they will actually terminate. Such proofs are important for the software developer, since they support the quality standards of his product. Nevertheless, termination of computer programs being an undecidable property, the termination of many methods remains unproved and such methods might hence be potentially non-terminating. A direct proof of their non-termination becomes desirable, since it exhibits an actual, typically unexpected behaviour of the program and often means that the non-terminating methods contain a bug. Currently, no system exists to prove the non-termination of Java bytecode methods, since research has mainly been focused on proofs of non-termination for logic programs [4,11,10,2,16,15] and term rewriting systems [21,5,24,22,23,9]. In the recent paper [7], the authors consider non-termination of C programs and [6,14] provide some techniques for testing C programs that detect errors such as program crashes, assertion violations, and non-termination. In [20], an approach to automatically check non-termination of imperative programs is introduced; it is based on the generation of invariants that are used to prove that some potential loops are never exited; the technique is experimented on a set of programs written in a fragment of Java and does not consider heap data structures. In this paper, we provide an example where our approach successfully proves the non-termination of a program where a data structure is defined.

This paper provides a first experimentation with the automatic derivation of non-termination proofs for Java bytecode programs. We start from our previous work on a tool Java-And-Form for the termination analysis of Java bytecode [19]. There, we translated the original Java bytecode program \( P \) into a constraint logic program \( P_{CLP} \) whose termination entails that of \( P \). Here, we show how, in those cases when the approximation of the bytecodes is exact, the non-termination of \( P_{CLP} \) entails that of \( P \). Hence, we use the same tool as in [19] to prove the non-termination of Java bytecode programs by exploiting previous work on Java bytecode programs [19] for non-termination analysis of logic programs [16]; namely, we prove the non-termination of \( P_{CLP} \) and hence infer, when possible, that of \( P \). Although these results are far from being a definite solution to the problem of non-termination analysis of Java bytecode programs, they represent a first step in that direction and highlight weaknesses of the current approach, that must be solved if non-termination analysis must be applied to real Java and Java bytecode software. Note that, while a notion of existential non-termination for C is considered in [7], we instead consider a notion of universal non-termination here for the CLP program derived from the Java bytecode program.

This paper also shows that our non-termination test effectively helps the termination test defined in [19]. Namely, we use our non-termination test to signal to the termination prover in [19] that some clauses in \( P_{CLP} \) diverge, so that it is useless to look for an (often expensive) termination proof for them. Note that this technique is applicable and profitable for all Java bytecode programs, also when the approximation of their bytecodes is not exact or when all their methods actually terminate. Our termination test is applied, indeed, to the CLP program, whose clauses might hence be potentially non-terminating. A direct proof of their non-termination becomes desirable, since it exhibits an actual, typically unexpected behaviour of the program and often means that the non-terminating methods contain a bug. Currently, no system exists to prove the non-termination of Java bytecode methods, since research has mainly been focused on proofs of non-termination for logic programs [4,11,10,2,16,15] and term rewriting systems [21,5,24,22,23,9]. In the recent paper [7], the authors consider non-termination of C programs and [6,14] provide some techniques for testing C programs that detect errors such as program crashes, assertion violations, and non-termination. In [20], an approach to automatically check non-termination of imperative programs is introduced; it is based on the generation of invariants that are used to prove that some potential loops are never exited; the technique is experimented on a set of programs written in a fragment of Java and does not consider heap data structures. In this paper, we provide an example where our approach successfully proves the non-termination of a program where a data structure is defined.

This paper provides a first experimentation with the automatic derivation of non-termination proofs for Java bytecode programs. We start from our previous work on a tool Java-And-Form for the termination analysis of Java bytecode [19]. There, we translated the original Java bytecode program \( P \) into a constraint logic program \( P_{CLP} \) whose termination entails that of \( P \). Here, we show how, in those cases when the approximation of the bytecodes is exact, the non-termination of \( P_{CLP} \) entails that of \( P \). Hence, we use the same tool as in [19] to prove the non-termination of Java bytecode programs by exploiting previous work on Java bytecode programs [19] for non-termination analysis of logic programs [16]; namely, we prove the non-termination of \( P_{CLP} \) and hence infer, when possible, that of \( P \). Although these results are far from being a definite solution to the problem of non-termination analysis of Java bytecode programs, they represent a first step in that direction and highlight weaknesses of the current approach, that must be solved if non-termination analysis must be applied to real Java and Java bytecode software. Note that, while a notion of existential non-termination for C is considered in [7], we instead consider a notion of universal non-termination here for the CLP program derived from the Java bytecode program.

This paper also shows that our non-termination test effectively helps the termination test defined in [19]. Namely, we use our non-termination test to signal to the termination prover in [19] that some clauses in \( P_{CLP} \) diverge, so that it is useless to look for an (often expensive) termination proof for them. Note that this technique is applicable and profitable for all Java bytecode programs, also when the approximation of their bytecodes is not exact or when all their methods actually terminate. Our termination test is applied, indeed, to the CLP program, whose clauses might

network. Were static fields left unmodified, each VM that loads class \( X \) would have its own copy of each field, leading to inconsistent data.

We use the concept of static singletons to maintain unique static data. The RuggedJ run-time library creates static objects on demand, and coordinates between nodes to guarantee that the network contains at most one instance of \( X_{\text{static}} \). Once some node has allocated the singleton, all other nodes will create \( X_{\text{static}_\text{stub}} \) objects as required. By managing static singletons through the run-time, we eliminate the need for a \( X_{\text{static}_\text{proxy}} \) class. Since rewritten code does not store references to the static singleton, we ensure that each node has a single reference to a given static singleton. Should the need arise to migrate a static singleton, we must update only this one reference.

### 3.4 Unmodifiable Classes

While RuggedJ can rewrite the majority of application classes as described in Section 3.1, there are a subset of application and Java standard library classes that it cannot. This limitation arises from the presence of native code. We cannot rewrite a method implemented using Java’s native interface, and such a method will not be aware of the presence of transformed classes. It may attempt to access fields or methods that we have modified or that are not available. We call these classes unmodifiable, and do not rewrite them. Tilevich and Smaragdakis define such classes to be those accessible by native code: classes that contain native methods, those passed to or returned from unmodifiable classes, the types of fields in unmodifiable classes, and superclasses of unmodifiable classes [20]. While it is theoretically possible for native code to access other (indeed, any) classes in the system, they found this heuristic to be sufficient for the realistic applications they examined.

Since we do not transform unmodifiable classes, we cannot distribute them. In practice, we find that the majority of unmodifiable classes exist within the Java standard libraries and are often closely tied to the underlying VM. This does not prove to be a great obstacle to the distribution of an application, since such classes would not move in any case. However, it is necessary that remote nodes be able to reference instances of unmodifiable classes. To this end, we generate wrappers for unmodifiable classes.

Figure 4 shows the classes we generate for an unmodifiable class. It is important to note that in this case, class \( Z \) is the original, unmodified class; we generate the wrapping classes as part of a reserved package to avoid naming conflicts with the original. Class \( \text{gen.Z}_\text{Local} \) acts as a wrapper around the original class \( Z \). It contains a reference to the instance of the unmodifiable class, with implementations of instance methods, each of which calls the appropriate method on the wrapped object. We generate \( \text{gen.Z}_\text{Stub} \) and

---

**Figure 4:** Wrapping unmodifiable classes

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Wrapping Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Z</td>
<td>class Z</td>
</tr>
<tr>
<td>class gen.Z</td>
<td>class gen.Z</td>
</tr>
<tr>
<td>class gen.Z_static</td>
<td>class gen.Z_static</td>
</tr>
<tr>
<td>interface gen.Z</td>
<td>interface gen.Z</td>
</tr>
<tr>
<td>class gen.Z_local</td>
<td>class gen.Z_local</td>
</tr>
<tr>
<td>class gen.Z_stub</td>
<td>class gen.Z_stub</td>
</tr>
</tbody>
</table>

---
3.5 Arrays

When distributing an application, we must transform not only objects but also arrays. To this end, we generate a set of interfaces and classes for every pair of array content and dimensionality used in the application. The interface contains get and set methods for the array content, as well as methods to perform standard operations such as getting the length or hash value for the array. For each pair of content and dimensionality used, we generate a wrapper class, as shown in Figure 5.

We generate a set of interfaces and classes for every pair of array content and dimensionality used in the application. The interface contains get and set methods for the array content, as well as methods to perform standard operations such as getting the length or hash value for the array. For each pair of content and dimensionality used, we generate a wrapper class, as shown in Figure 5.
We generate the classes for multi-dimensional arrays in the same way as for single dimension arrays (we represent a two-dimensional array of $Y$ by interface $\text{Array of } Y^2$ and so forth), with the local class containing a wrapped array. The wrapped array is always one-dimensional, so $\text{Array of } Y^2\text{ local}$ contains an array of $\text{Array of } Y^2$ objects. As well as simplifying the implementation of arrays, this design allows us to spread large multi-dimensional arrays across multiple nodes.

3.6 Hand-Coded Classes

A final, small, subset of classes within RuggedJ are hand-written and loaded unmodified into the Java VM. These are classes that require specific, customized implementations within the RuggedJ network. For example, $\text{java.lang.System}$ contains several methods for which we define special semantics: we must redirect all references to $\text{System.out}$ to the head node, rather than to the local machine. Since performing such one-off transformations would be laborious and would complicate the transformation framework, we prefer instead simply to load a hand-coded version of these classes.

4 Implementation

Beyond generating new classes, implementation of the RuggedJ object model requires widespread modification to application bytecode. In this section we describe some of the more interesting features of the rewriting process.

4.1 Bytecode Rewriting Tools

When implementing RuggedJ, the first decision we needed to make was the level at which to rewrite. High-level tools such as AspectJ [10] and MetaAspectJ [8] would allow us to specify RuggedJ’s transformations in Java source code. While this is adequate to add code to a method, more complex transformations would require an additional tool. A more flexible approach is that of Javassist [3, 4], which allows one to specify transformed code in Java syntax, which it compiles with a custom compiler. This offers a lower-level interface to rewriting. However, we found that its on-demand compilation approach made whole-program modification difficult. Ultimately, we found that ASM [1] supports a good balance of direct access to method bytecode while hiding awkward details such as management of constant pools and the selection of instructions with hard-coded local variable slots. These two abstractions vastly simplified the design of transformations and generated bytecode, making ASM more useful to us than the similarly-featured BCEL [5].

4.2 Transforming Method Bodies

Of the classes we generate for a given application class, only the local and static local versions contain copied bytecode. We generate all other classes from scratch. Thus, we apply the following transformations only to the bodies of local and static local classes.

Instance Method Invocation: We must refer to all transformed objects in RuggedJ by interface rather than class type, allowing us to vary the implementation of a class among proxy, local, or stub transparently to the calling code. This clearly requires modification to method call sites, transforming invokevirtual bytecodes to invokeinterface.
We need a more complicated rewrite in the case of invokespecial bytecodes, used to call private methods, constructors, or superclass methods. We can call private methods in the same way as regular methods (for the sake of simplicity, we modify all methods to be public; the original Java compilation enforced the access controls). However, we cannot call constructors or superclass methods through an interface. We must invoke a constructor upon the appropriate class; we describe this process in Section 4.4. Superclass invocations must specify the superclass type upon which to invoke a method (in case a subclass has overridden the method). This does not present a problem since we know that the code we are modifying is within a local class, the superclass of which we also know.

Instance Field Accesses: We must rewrite accesses to instance fields, since direct access to a field assumes that an object is local. To this end, we replace every instance field access by a call to the appropriate get or set method in the interface.

This policy obviously adds an unnecessary level of indirection when the accessed field belongs to the accessing object. A more subtle problem exists, however, that necessitates special handling of such accesses. Under the Java VM specification, the only operation that may occur in a constructor before the invocation of a superclass constructor is the initialization of a field in the local object. Rewriting such a field invocation to a method call would cause a verification error, since a method call cannot precede the superclass constructor call. We can detect cases where a field access occurs on the accessing object using a simple flow analysis, as we describe in Section 4.5.

Static Method Bodies: As we discussed in Section 3.3, we transform static fields and methods within the static local class to be members of the static singleton object. While transforming fields is straightforward, we must rewrite static method bodies to function as instance methods. The first local variable slot in an instance method is reserved for the this pointer, referring to the object upon which the method is invoked. Static methods are not invoked on any object, and so do not have a this pointer. Thus, when converting a static method to an instance method we must to update all local variable references to allow for the new reference.

This transformation can cause major changes to the bytecode sequence of a method. Not only does it change the parameters to local variable bytecodes, but the bytecodes themselves may change. For example, the aload_3 bytecode operates as an aload with a parameter of 3. Incrementing the local variable slot upon which this bytecode operates would require replacing the aload_3 bytecode (a one-byte instruction) by an aload with an argument of 4 (a two-byte instruction). This will affect the offsets of future bytecodes, and will require updates to jump instructions, exception handling blocks, and so on. Fortunately, a bytecode rewriting toolkit such as ASM abstracts away most of these details.

4.3 Accessing Static Singletons

As in the case of instance field accesses and method invocations, we must rewrite static accesses. However, the presence of static singletons makes the process somewhat more complex. First, the RuggedJ run-time library must locate the appropriate static singleton by looking it up in a hash table of static objects. If the required singleton is unavailable, the run-time library must first determine whether a singleton exists on another node and, failing that, create one. This involves coordination with the other nodes in the network to find an existing singleton, or synchronization with the head node to avoid two nodes...
null with three predicates $\leq, <,$ and $=.$ This class is decidable according to the classical result by Bernays, Schönhage [see e.g. [9]]. Interestingly enough, this class is more powerful and the formulae could take up, roughly, the form of the $\forall x \exists y \forall z \forall w \forall v \forall u \bigl( (x+y) = (z+w) \land (v+w) = (u+u) \bigr).$ The safety environment and it is the abstract stack. In this way, we would not need to rely on some external source to supply the arrays and the decision procedure for the first-order logic would infer the typing for the program. However, one cannot quantify in BML so that the quantification ranges over different anchor formulae. Therefore, this approach is not available directly.

This obstacle could be overcome with the help of the observation that the satisfiability of the formulae does not depend on the values of the source code variables and the control flow of the program. This lets us to store the conjunction of all the formulae in the method precondition $R_m$. That, however, would make the implementation of the declassification more involved. In this framework, the declassification must be implemented by a modification of the formulae themselves instead of the modification of the data they operate on.

**Future work** Currently, it is rather difficult to present a single succinct example of how the translation works as the result of the translation is rather complicated. At the moment, we need to transform the inferences in the information flow type system to BML using the translation under the development.

**Acknowledgement** The authors thank the anonymous referees for all the comments which helped to improve the paper.

**References**

the objects of a class over the network: the location of the object will depend on the run-time distribution pattern. This option defers the decision of whether to allocate a local or stub class until run time.

Each of these options causes the rewriting class loader to replace the allocation site with a different bytecode sequence. In the case of a local allocation, the bytecode sequence simply creates a new local object, with or without a proxy. The remote allocation sequence involves a call to the run-time library to determine the node upon which to create the object, then a remote creation request and creation of a stub object and proxy. Finally, the dynamic allocation option generates both sets of bytecode, with a call to the run-time library to determine which to execute.

A final complication when rewriting an allocation site is that of calling the appropriate constructor. The constructor call for an object can be an arbitrary distance from the new bytecode that creates the object to pass to the constructor, since there may be an arbitrary number of operations to compute the arguments to the constructor. There may even be other constructor calls between the two bytecodes, since the arguments to the constructor may require creation of new objects. We take advantage of the fact that every new bytecode has exactly one constructor call, and so we can match a new bytecode with its constructor call using a simple stack-based scanning technique. We can forward through the bytecode stream pushing any new bytecodes, and popping them when we encounter constructor calls. The final constructor we encounter therefore belongs to the original new bytecode.

4.5 Flow Analysis

The vast majority of bytecode modifications in RuggedJ are context-independent; their implementations do not require knowledge of the method as a whole. Aside from the method scanning required to locate constructors mentioned above, there exist two cases for which we need to analyze the method body.

The first concerns operations on arrays. As discussed in Section 3.5, we replace all arrays in a RuggedJ network with wrapping objects. This presents problems during the rewriting phase since, unlike most bytecodes that operate over references, array operations (aload, astore, arraylength, etc) do not encode type information. One can determine the type of the array reference and return value only by modeling the run-time stack. Since we rewrite these bytecodes to standard invokevirtual method calls, we need to know both the type and dimensionality of the array upon which to invoke the method. Finding this information using a standard bytecode flow analysis of types of objects that tags each array bytecode with the type of array currently on top of the stack.

The second flow analysis we require is to track the this pointer in instance methods. As mentioned in Section 4.2, the RuggedJ rewriting class loader must differentiate between field accesses on the current object and those on others. Since we know that the this pointer exists in local array slot 0, we can track any references that start life with an astore bytecode, determining them to be references to the current object.

This analysis is, by its nature, conservative. It can produce a false negative when, for example, code passes a reference to a method that then returns something of the same type. The return value could be the original reference or a different object. This conservatism is not a problem since we employ the analysis mostly for optimization, so missing a reference does not violate correctness.

The only occasion where we rely on this analysis is where conflict with TR(P, kobs, ft, Γ, cdr, se, st). If P′ verifies correctly then P satisfies the non-interference property.

Proof:
We present here a sketch of the proof only. When the specification extension does not conflict with the translation TR(P, kobs, ft, Γ, cdr, se, st) then the values of all the variables used in the translation are the same. In this light, the logical values of the formulas are the same as in case there are no additional specifications. Hence we obtain the non-interference for P.

5 Discussion of the solution

Declassification: Our translation modifies the local assertion table Qm so that each typing rule is checked right before the instruction instance it concerns. Note that the logic, as presented in Sect. 2, allows to change the state of the ghost variables by means of the local instruction table Im. This enables an easy method to declassify information by means of the assignment to a ghost variable. Usually, the declassification should occur when a value on a high security level is to be released to a lower security level. This can be done at a program point i should be stored in a low level field. The current rules prevent this, but they exploit the information stored in entries of gsgn and gset arrays that correspond to i. We can exploit a new instruction in Im, (i) to change gsgn and gset right before the instruction that requires the declassification and revert it back right the next one. In this way we obtain in a clear declassification management mechanism — declassification is present when the set instructions manipulate the mentioned above arrays.

In fact, the presented method can also be applied to many other type systems which are information flow sensitive — the flow of the information is simply traced by the ghost variables. In essence, the practice of using the ghost variables in programs specified in JML is in many cases such that they serve as a method to provide an ad hoc information flow typing system.

Finite range of levels: The original order used in the information flow type system has not been restricted to be finite. Our translation relies crucially on the fact that the order is finite. In practice, however, it is very difficult to check the non-interference in case of non-finite infinite policies — in particular such policies should be effectively enumerable and thus the checking that a policy is fulfilled becomes rather an algorithm verification task than static checking.

Design choices: The primary goal of the design choices we took here was to find a way to express a system which ensures the non-interference property in terms of the BML formulae. The main challenge here was to connect the flow of the data with the first order formulae available in the language. We decided to simulate in ghost variables the operation of the type checking.

Another possibility would be to use ghost variables to simulate an alternative operation of the program in a flavor similar to the approaches [8,6,13]. However, the operation on the variables would use the control flow of the original program and it is not clear if it is possible to express the non-interference in this way.

The formulae we generate in our approach fall easily within the class of ∀∗ for.
Suppose that the annotated program $TR(P, k_{obs}, ft, s, e, s, st)$ verifies correctly. We want to show that $P$ is typable, i.e., that every method $m$ in $P$ is typable with respect to every signature in PoliTyp($m$). We need to verify that the condition in Def. 1 is fulfilled. Let $s$ be a signature corresponding to security level $s$. We take $st$ and $s$ as above. $st[0] = e$ is guaranteed by the way $gst$ is initialised. Since $P$ verifies correctly the formula $N(i)(s)$ holds for every $i$. The conditions (1)-(2) from Def. 1 are guaranteed by the fact that all the typing rules are faithfully modelled in the logic. Let us see it for $inst(i) = ifeq$. In other cases the proof is similar.

$true$ as an instruction at $i$ is $ifeq$, we must only ensure condition (i) of Def. 1. In case of the $ifeq$ instruction, the body of the formula $N(i)(s)$ is $(i \leftarrow j + 1)$.

$(Reg_{inst}^{eq}(j))$. This formula ensures that in case $j = j_0$ or $j = i + 1$ the formula $Reg_{inst}^{eq}(j_0)$ holds. This ensures that the check of the premises of the rule from Figure 1 takes place indeed for the instruction $ifeq$. Then, as the first (forall) subformula of $Reg_{inst}^{eq}(j_0)$ holds, we obtain $\forall j' \in region(i, 0), k < set(j')$ as $k$ is identified with $gst[s][i][\text{ctr}]$. The second and the third (forall) subformulas of $Reg_{inst}^{eq}(j_0)$ ensure that $lift(st) \subseteq st(j)$ (where $j = i + 1$ or $j_0$).

Suppose that the program $P$ with the policy $(k_{obs}, ft, s, e)$ is typable with $st$ and $s$. We have to ensure that each $QN_{inst}(i)$, for $i$ being a label in the method $m$. An $Ql(st, s)$ is true, it is enough to check that each $N(i)(j)$ holds for $j \in S$. Each of the $N(i)(s)$ has similar structure presented in (2). It is enough to show that one of the corresponding $Reg_{inst}^{eq}(j_0)$ holds in case $i \leftarrow j$ (or in case $i \leftarrow j_0$).

As the method is typable, we know that $\Gamma, ft, region_{obs}, s, e, s, i \leftarrow st(i) \Rightarrow s$ (or $\Gamma, ft, region_{obs}, s, e, s, i \leftarrow st(i)$) can be inferred. This is done with one of the rules, say $\text{ldh}$. Now, we have to make sure that the corresponding translation formula $Reg_{inst}^{eq}(j_0)$ holds. We show this in case $inst(i) = ifeq$.

- The first subformula of (3) holds as the typing rule guarantees that the property $\forall j' \in region(i, 0), k < set(j')$ holds,
- the second subformula of (3) holds as the typability requires that $lift(st) \subseteq st(j)$, where $st$ is $st(i)$ without its top element,
- the third subformula of (3) holds as the $st(j)$ is not determined for indices greater than the top of the operand stack.

This finishes the proof in this case. The cases of other instructions are similar. $\blacksquare$

### 4.1 Proof of stability

Theorem 4.2 below states that we can safely extend the specifications so that the non-interference property is preserved. More precisely, it allows to mix the specifications that result from our translation with specifications that come from other sources (e.g., are written by hand).

**Definition 2 (specifications in conflict)**

We say that specifications are in conflict with the translation $TR$ whenever any element of the ghost arrays or variables defined in Sect. 3 is set.

**Theorem 4.2 (stability)**

Let $P'$ be a specification extension of $TR(P, k_{obs}, ft, s, e, s, st)$ that does not

---

(⇒) Suppose that the annotated program $TR(P, k_{obs}, ft, s, e, s, st)$ verifies correctly. We want to show that $P$ is typable, i.e., that every method $m$ in $P$ is typable with respect to every signature in PoliTyp($m$). We need to verify that the condition in Def. 1 is fulfilled. Let $s$ be a signature corresponding to security level $s$. We take $st$ and $s$ as above. $st[0] = e$ is guaranteed by the way $gst$ is initialized. Since $P$ verifies correctly the formula $N(i)(s)$ holds for every $i$. The conditions (1)-(2) from Def. 1 are guaranteed by the fact that all the typing rules are faithfully modeled in the logic. Let us see it for $inst(i) = ifeq$. In other cases the proof is similar.

(⇐) Suppose that the annotated program $TR(P, k_{obs}, ft, s, e, s, st)$ verifies correctly. We want to show that $P$ is typable, i.e., that every method $m$ in $P$ is typable with respect to every signature in PoliTyp($m$). We need to verify that the condition in Def. 1 is fulfilled. Let $s$ be a signature corresponding to security level $s$. We take $st$ and $s$ as above. $st[0] = e$ is guaranteed by the way $gst$ is initialized. Since $P$ verifies correctly the formula $N(i)(s)$ holds for every $i$. The conditions (1)-(2) from Def. 1 are guaranteed by the fact that all the typing rules are faithfully modeled in the logic. Let us see it for $inst(i) = ifeq$. In other cases the proof is similar.

In case of the $ifeq$ instruction, the body of the formula $N(i)(s)$ is $(i \leftarrow j_0)$.

$(Reg_{inst}^{eq}(j_0))$. This formula ensures that in case $j = j_0$ or $j = i + 1$ the formula $Reg_{inst}^{eq}(j_0)$ holds. This ensures that the check of the premises of the rule from Figure 1 takes place indeed for the instruction $ifeq$. Then, as the first (forall) subformula of $Reg_{inst}^{eq}(j_0)$ holds, we obtain $\forall j' \in region(i, 0), k < set(j')$ as $k$ is identified with $gst[s][i][\text{ctr}]$. The second and the third (forall) subformulas of $Reg_{inst}^{eq}(j_0)$ ensure that $lift(st) \subseteq st(j)$ (where $j = i + 1$ or $j_0$).

(⇐) Suppose that the program $P$ with the policy $(k_{obs}, ft, s, e)$ is typable with $st$ and $s$. We have to ensure that each $QN_{inst}(i)$, for $i$ being a label in the method $m$. An $Ql(st, s)$ is true, it is enough to check that each $N(i)(j)$ holds for $j \in S$. Each of the $N(i)(s)$ has similar structure presented in (2). It is enough to show that one of the corresponding $Reg_{inst}^{eq}(j_0)$ holds in case $i \leftarrow j$ (or in case $i \leftarrow j_0$).

As the method is typable, we know that $\Gamma, ft, region_{obs}, s, e, s, i \leftarrow st(i) \Rightarrow s$ (or $\Gamma, ft, region_{obs}, s, e, s, i \leftarrow st(i)$) can be inferred. This is done with one of the rules, say $\text{ldh}$. Now, we have to make sure that the corresponding translation formula $Reg_{inst}^{eq}(j_0)$ holds. We show this in case $inst(i) = ifeq$.

- The first subformula of (3) holds as the typing rule guarantees that the property $\forall j' \in region(i, 0), k < set(j')$ holds,
- the second subformula of (3) holds as the typability requires that $lift(st) \subseteq st(j)$, where $st$ is $st(i)$ without its top element,
- the third subformula of (3) holds as the $st(j)$ is not determined for indices greater than the top of the operand stack.

This finishes the proof in this case. The cases of other instructions are similar. $\blacksquare$

### 4.1 Proof of stability

Theorem 4.2 below states that we can safely extend the specifications so that the non-interference property is preserved. More precisely, it allows to mix the specifications that result from our translation with specifications that come from other sources (e.g., are written by hand).

**Definition 2 (specifications in conflict)**

We say that specifications are in conflict with the translation $TR$ whenever any element of the ghost arrays or variables defined in Sect. 3 is set.

**Theorem 4.2 (stability)**

Let $P'$ be a specification extension of $TR(P, k_{obs}, ft, s, e, s, st)$ that does not

---

A field initialization occurs prior to the super-constructor call in a constructor. However, since the only field initializations that may occur before that call are to the local object, the analysis will always be accurate in this case.

### 4.6 Uncooperative Code

As with most large-scale automatic application transformation systems, RuggedJ cannot guarantee correctness in all cases. There are certain corner cases where an adversarial programmer can foil the rewriting system into producing incorrect results. However, we are confident that such cases are rare under normal circumstances.

The most apparent area in which our rewrites might lead to errors is reflection. An application developer generally has more knowledge of the run-time properties of objects in an application, and could use Java’s reflection system to perform operations on a class that may not be possible in the rewritten system. With that said, we do take measures to avoid this by intercepting reflective calls and updating arguments or types to fit within the RuggedJ system, allowing most common usages of reflection to operate within our system.

We also do not support applications that define their own custom class loaders. Since RuggedJ uses a rewriting class loader, we cannot integrate the operations that may be performed by an application’s own class loading system.

Finally, we are aware of several ways in which native code could produce incorrect results within RuggedJ. The heuristics discussed in Section 3.4 allow our system to accommodate most native code, but the Java Native Interface allows native code virtually limitless access to the VM. By allocating or invoking methods on arbitrary objects a native method can perform operations that are incompatible with RuggedJ’s transformations. This problem will most likely arise in a non-adversarial application though use of static singletons.

The Java Native Interface `CallStaticTypeMethod` methods allow native code to invoke static methods of arbitrary classes. Reflectively invoking a static method of a class that requires a static singleton will result in the call failing in RuggedJ. However, allowing for arbitrary static method calls would mean that no class could have a static singleton and so could only be accessed from a single node, making distribution impossible.

### 5 Related Work

The system that most closely resembles RuggedJ is J-Orchestra [18]. Indeed, J-Orchestra influenced many of RuggedJ’s original design decisions. However, J-Orchestra’s fundamental goal is different from RuggedJ’s. J-Orchestra aims for “resource-driven distribution,” where one shares an application between a small set of machines with specific capabilities. For example, a transformed system may perform calculations on a back-end server, while displaying its user interface on a PDA. This differs from RuggedJ’s goal of distributing an application across a cluster of machines, taking advantage of additional hardware to exploit parallelism. The design of each system reflects these differing objectives.

The major difference between the two systems is that RuggedJ performs dynamically many functions that J-Orchestra performs statically. J-Orchestra determines a partitioning ahead of time for a given network configuration. Guided by a whole-program analysis, a user determines which classes should have their instances allocated on each network location. This approach works well for J-Orchestra’s usage, since it targets small clusters with
clear roles for each machine. However, RuggedJ performs this partitioning at run time using an application-specific partitioning plug-in to decide dynamically upon the location of remote objects. Similarly, one can see the static/dynamic difference in the way in which J-Orchestra rewrites application code. It transforms classes ahead of time, generating proxies and remote representations as Java source that one then compiles, producing a jar file for each network location. This is in contrast to our approach of rewriting at class load-time, which gives us the ability to generate bytecode tuned to the RuggedJ network upon which the application is running, and removes the system's dependence on an external compiler.

Another consequence of J-Orchestra’s ahead-of-time partitioning strategy is that it makes all partitioning decisions on a per-class basis. In contrast, RuggedJ’s dynamic partitioning system allows per-instance decisions, allowing us to allocate instances of a given class on arbitrary nodes within the network. Not only does this let us take advantage of current network conditions that cannot be predicted ahead of time, but it also allows us to perform load-balancing by distributing key objects of a given class across the network.

Finally, there are differences in the object model implemented by each system that we feel allow RuggedJ more flexibility when executing large applications. In J-Orchestra, the fundamental class for objects that code may reference remotely is the proxy, while in RuggedJ it is the interface. Rewritten bytecode in J-Orchestra refers to proxies rather than interfaces, removing the ability to elide proxies for objects that are known to be either local or remote. Additionally, J-Orchestra’s approach to arrays differs in that it considers arrays of a given type but of different dimensionality to be related, while RuggedJ considers an array type to consist of both a base type and dimension, allowing for a more flexible partitioning scheme.

There exist several other projects that seek to simplify the distribution of Java. Space limitations prevent us from discussing these systems in detail, but none follow the same approach as RuggedJ.

Terracotta [17] is an open-source JVM-level clustering framework that uses bytecode rewriting techniques to generate a distributed Java application without the requirement to code to a specific API. The Terracotta approach is superficially similar to that taken by RuggedJ, but there are several fundamental differences: Terracotta requires that the application developer label “root” references with altered semantics through which one can reach shared objects, while RuggedJ considers all objects as potentially reachable from remote nodes. Additionally, Terracotta is heavily based upon a central server node, which manages the canonical versions of all shared objects. We maintain canonical versions of objects throughout the cluster.

Addistant [16] uses bytecode transformation to distribute legacy code, but does not aim to distribute large parts of the application. AIDE [14] uses a modified JVM to offload execution from portable devices to servers, whereas RuggedJ runs on unmodified VMs.

JavaParty [7, 15], Javanaise [6], Do! [11, 12] and Java// [2] each provide language-level features to Java that simplify distributed programming, while RuggedJ performs its transformation at the bytecode level without modification to the original source.

6 Conclusion

Whole-program transformation is a powerful technique that allows one to add substantive new functionality to an existing off-the-shelf application. In this paper we have presented
signature of the method (recall that for each security level \( s \) there is a separate security signature \( \Gamma_{N(m)}[s] \)) applicable in the situation when the object on which \( m \) is called has the security level \( s \), every label \( i \), every exception \( e \), and every \( j \), such that \( i \rightarrow^e j \) (or \( i \rightarrow^e j \)) some properties hold. For every \( i \) these properties are expressed by formulae \( N(i)(s) \). We define \( Q_N(i) \), the local annotation table extended with the non-interference checking, as

\[
Q_N(i) = \lambda 0 \in \text{State} \quad \lambda c \in \text{State}.
\]

This formula expresses a new assert before the instruction at \( i \) that states that the old assert must hold together with all the security guarding formulae which ensure (together with all the formulae for other instructions) that the security signatures of \( m \) are satisfied.

The security guarding formula \( N(i)(s) \) has similar form; it is

\[
(\forall \text{all int } e, j \quad 0 \leq e \leq \text{exMax} \quad 0 \leq j \leq j < \text{lm} \quad (i \rightarrow^e j) \Rightarrow (\text{Reg}_{\text{inst}(i)}(p_1) \land \ldots \land \text{Reg}_{\text{inst}(i)}(p_k) \land c \land c \\ (i \rightarrow^e j) \Rightarrow (\text{Reg}_{\text{inst}(i)}(p_1) \land \ldots \land \text{Reg}_{\text{inst}(i)}(p_k) \land c))^{19}
\]

where \( \text{inst}(i) \) is the instruction at the label \( i \) in the method body \( m \). Note also that \( i \rightarrow^e j \) (as well as \( i \rightarrow^e j \)) static information which can be defined directly as a subformula. For example, in the most typical case when the control flow moves to the next instruction, the formula is of the form \( j = j + 1 \land c \land e = \text{maxEx}^{10} \). Here, the condition that \( \text{maxEx} \) equals \( c \) enforces that we consider a normal step. In the case when the method has an exception handler of \( c \) at the point \( j \) we define \( i \rightarrow^e j \) to be \( j = j + 1 \land c \land e = e \). As the \( \text{ifeq} \) \( j \) has two successors, but one rule handles the instruction in the type system, the premise of the implication is \( (j = j + 1 \quad \text{or } j = j) \land e = \text{maxEx} \) in this case.

Every \( \text{Reg}_{\text{inst}(i)}(p_k) \) for \( i = 1, \ldots, k \) or \( e \) for \( e = k + 1, \ldots, k' \) corresponds to one of possibly applicable typing rules for instruction \( \text{inst}(i) \) in case \( i \rightarrow^e j \) (or \( i \rightarrow^e j \)). The type system considers an instruction to be correct when at least one of the rules can be successfully satisfied. Therefore, the formula \( \text{Reg}_{\text{inst}(i)}(p_k) \) are combined as an alternative. Let us point out that the vectors \( p_k \) are parameters of the instruction \( \text{inst} \). For instance, there is one rule for \( \text{ifeq} \) and \( \text{Reg}_{\text{inst}(i)}(j) \), corresponding to

---

8 We add this information only in case the instruction may throw an exception.

9 Note that this solution can be formulated as \( \text{forall } i \) in the course of the translation and therefore is not a part of the formula syntax.


\[
ghost boolean \ [lm] \ [maxEx] \ class Analysis = \{ \ \{ c_0, 0, \ldots, c_0, maxEx - 1 \} , \ldots , \{ \ldots \} \} ;
\]

where

\[
e_{i,j} = \begin{cases} 
true \text{ when } classAnalysis(i) \text{ says that the exception } j \text{ can be thrown in } m \text{ at } i, \\
false \text{ otherwise.}
\end{cases}
\]

\[
ghost boolean \ [lm] \ [maxEx + 1] \ [lm] \ region = \{ \\
\{ \{ h_{0,0}, \ldots, h_{0,0}, maxEx - 1 \} , \ldots , \{ h_{0,0,0}, \ldots, h_{0,0,0}, maxEx - 1 \} \} , \ldots , \\
\{ h_{0,1,0}, \ldots, h_{0,1,0}, maxEx - 1 \} , \ldots , \{ h_{0,1,0,0}, \ldots, h_{0,1,0,0}, maxEx - 1 \} \} ;
\]

where

\[
e_{i,j,k} = \begin{cases} 
true \text{ when } k \in \text{ region}_m(i, e) \text{ and the exception corresponding to } e \text{ is } j, \\
false \text{ otherwise.}
\end{cases}
\]

Note that we use the index maxEx on the second coordinate to encode the region information for the normal execution.

\[
ghost int \ [lm] \ [maxEx] \ [lm] \ [Handler] = \{ \\
\{ h_{0,0}, \ldots, h_{0,0}, maxEx - 1 \} , \ldots , \{ h_{0,1,0}, \ldots, h_{0,1,0}, maxEx - 1 \} \} ;
\]

where \( h_{i,j} = \text{Handler}_m(i, e) \) with \( e \) corresponding to the exception number \( j \).

**Type system data** As noted below Def. 1, we may assume that \( se \) and \( st \) are indexed with security levels from \( S \). We use the notation \( se \), and \( st \), for \( i \in S \) to refer to the elements of the indexed families.

\[
ghost int \ [max] \ [lm] \ [se] = \{ \\
\{ h_{0,0}, \ldots, h_{0,0}, maxEx - 1 \} , \ldots , \{ h_{maxS - 1,0,0}, \ldots, h_{maxS - 1,0,0}, maxEx - 1 \} \} ;
\]

where \( e_{i,j} = se(j) \).

\[
ghost int \ [max] \ [lm] \ [macStack] \ [get] = \{ \\
\{ h_{0,0,0}, \ldots, h_{0,0,0}, maxEx - 1 \} , \ldots , \{ h_{maxS - 1,0,0,0}, \ldots, h_{maxS - 1,0,0,0}, maxEx - 1 \} \} ,
\]

where \( t_{i,j,k} = n \) whenever \( (k + 1)\text{-st element of the sequence } st(i,j) \text{ is } n \). Note that the function \( st \) gives security levels for the stack positions so that the length of each \( st(i,j) \) is less than the maximal stack length \( maxStack \). We also assume that for each \( i \), \( j \) the elements of \( get[i][0][0][j] \) are zero which corresponds to the fact that the operand stack at the beginning of a method is empty.

**Translating the rules** Once we have all the annotations above, we may encode the capability property from Def. 1. We do it for each method in separately and we decide to use the local annotation table \( Q_m \) (as in [12, Chapter 3]).

\( Q_m \) is a finite partial map which for a program label \( i \) in \( m \) gives an assertion \( Q_i(s_0, s) \). If the point \( i \) in \( m \) is annotated with \( Q_m \) then \( Q_i(s_0, s) \) is supposed to hold in every state \( s \) at \( i \) during any execution of \( m \) with the initial state \( s_0 \) satisfying \( f_m(s_0) \) (i.e. the precondition of the method). Intuitively, \( Q_i(s_0, s) \) provides the content of the assert statement right before the instruction with the label \( i \).

Let us describe how to extend a given specification \( Q_m \) so that it ensures the non-interference property. According to Def. 1, we need to state that for every security
• Im the maximal label of the method m.
• maxStack the maximal height of the stack in the execution of method m.

These values are computed during the translation process.

Security requirements and program structure Security level of fields can be stored in ghost fields in the corresponding class. For each class field f (both static and instance) we define:

```
public final ghost int gargsN[m] = n(M);
```

this enables access to the security levels of f. The domain of Γ, excAnalysis, nblocs, nblags, nblargs is the set of methods names. They are stored in ghost fields of the class where the given method name appears highest in the class hierarchy. The other data are stored as local ghost variables of the actual methods.

In the definitions below, we use a fixed correspondence between the exception types and the natural numbers 0, ..., maxEx − 1. For each method m (both static and instance) with the identifier N(m), we define a set of ghost variables. These variables will be used as constants; they will never be changed. The initial values of all the ghost variables we use here are defined to correspond directly to the values of real variables/functions.

```
public static final ghost int nblargsN[m] = nblargsN(m);
public static final ghost int nbllocsN[m] = nbllocsN(m);
public static final ghost boolean maxex = excAnalysisN[m] == maxEx;
```

The last two definitions make use of additional values defined below. The information contained in excAnalysisN[m] is defined with the use of:

```
v[i] = \begin{cases} 
true & \text{when excAnalysisN[m]} \text{ says that the exception i is thrown in m}, \\
false & \text{otherwise}.
\end{cases}
```

The security signature \( \Gamma_{m}[i] = k_p \rightarrow k_v \) allows us to give the values for \( s_{i,j} \):

```
s_{i,j} = \begin{cases} 
kp[i] & j < kp[i] \\
kp[i] & j = kp[i] \\
k_p[i] - \left( k_p[i] - 1 \right) & j > kp[i].
\end{cases}
```

This definition allows us to explain the meaning of gargsN[i][j] in the following way. For a given security level \( i \), gargsN[0][i] is the security level of the object that calls the method \( m \). gargsN[1][i][nblargsN[1][N(m)]] are security levels of parameters and local variables (note that nblargsN[1][N(m)] = kp[i] − 1), the value gargsN[1][i][nblargsN[1][N(m)] + 1] is the level of loop operations, the value gargsN[1][i][nblargsN[1][N(m)] + 2] is the level of a normal return value and gargsN[1][i][nblargsN[1][N(m)] + 1 + gargsN[1][i][nblargsN[1][N(m)] + 1] + maxEx − 1] are the security levels in which corresponding exceptions might be propagated (note that maxEx ≥ kp[i] − 1).

We define also local ghost variables associated with the method mc:

```
public static final ghost int gargsL[0] = gargsL[0];
```

Virtual-Machine Abstraction and Optimization Techniques

Stefan Brunthaler
Institute of Computer Languages
Vienna University of Technology
Vienna, Austria

Abstract
Many users and companies alike feel uncomfortable with execution performance of interpreters, often also dismissing their use for specific projects. Specifically, virtual machines whose abstraction level is higher than that of the native machine they run on, have performance issues. Several common existing optimization techniques fail to deliver their full potential on each machine. This paper presents an exploration for this situation and provides hints on possible alternative optimization techniques, which could very well provide substantially higher speedups.

Keywords: Interpreter, Virtual-Machine Abstraction, Optimization Techniques.

1 Motivation
1000 : 10 = 1. These are the slowdown-ratios of an inefficient interpreter, when compared to an efficient interpreter, and finally to an optimizing native code compiler. Many interpreters were not conceived with any specific performance goals in mind, but rather striving for other goals of interpreters, among them portability; ease of implementation. This also means that there is a large benefit in optimizing an interpreter before taking the necessary steps to convert the tool chain to a compiler. There are common optimization techniques for interpreters, e.g. threaded code [2],[5],[8], superinstructions [9], and switching to a register based architecture [20]. The mentioned body of work provides helpful analyses and in-depth treatment of performance characteristics, implementation.

Those optimization techniques, however, have one thing in common: their basic assumption is that interpretation’s most costly operation is instruction dispatch, i.e., in getting from one bytecode instruction to its successor. While this assumption is certainly true for the interpreters of languages analyzed in the corresponding papers, e.g., Forth, Java, and OCaml, our recent results indicate that it is specifically

1 Email: brunthaler@complang.tuwien.ac.at
not true for the interpreter of the Python programming language. We find that this correlates with a difference in the virtual machine abstraction levels between their corresponding interpreters.

In virtual machines where the abstraction level is very low, i.e., essentially a 1:1 correspondence between bytecode and native machine code, the basic assumption of dispatch being the most costly operation within an interpreter is valid. Members of this class are the interpreters of Forth, Java, and OCaml, among others. Contrary to those, interpreters that provide a high abstraction level do not support this assumption. The interpreters of Python, Perl, and Ruby belong here. Additionally, we analyze the interpreter of Lua, which is somewhere in between both classes.

In their conclusion, Piumarta and Riccardi [15] suppose the following: “The expected benefits of our technique are related to the average semantic content of a bytecode. We would expect languages such as Tcl and Perl, which have relatively high-level opcodes, to benefit less from macroization. Interpreters with a more RISC-like opcode set will benefit more — since the cost of dispatch is more significant when compared to the cost of executing the body of each bytecode.” Our work shows, whether their expectations turn out to be correct, and we make explicit what is only implicitly indicated by their remark. Specifically we contribute:

- We categorize some virtual machines according to their abstraction level. We show which characteristics we consider for classifying interpreters, and provide hints regarding other programming languages than those discussed.
- We subject optimization techniques to that categorization and analyze their potential benefits with respect to their class. This serves as a guideline for a) implementers, which can select a set of suitable optimization techniques for their interpreter, and b) researchers which can categorize other optimization techniques according to our classification.

2 Categorization of Interpreters

To get a big picture on the execution profile of the Python interpreter, we collected 9 million samples of instruction execution times running the pystone benchmark on a modified version of the Python 3.0rc1 interpreter, which samples CPU cycles. We run Operation Execution, Dispatch and Whole Loop costs. Operation Execution contains all cycle costs for the first machine instruction in operation implementation until the last. Dispatch costs contain the number of cycles spent for getting from one operation to another, e.g. in a switch-statement from one case to another. Python’s interpreter, however, does not dispatch directly from one byte-code to another, but maintains some common code section which is conditionally executed before dispatching to the next instruction. To account for that special case, we measured so called Whole Loop costs, which measure the CPU cycles from the first and last instructions within the main loop.

Based on extensive previous work, [2,5,8,9,30], we expected that instruction dispatch would be the most costly interpreter activity for Python’s virtual machine, too. Figure 1 shows our results obtained by examining CPU cycles for the Python 3.0rc1 interpreter, running on a Pentium 4, 3 GHz, with Xubuntu 8.04.

summary of the relevant BML syntax.

We use the modifier ghost to indicate that a particular variable is not a program variable, but a specification variable. Other Java type modifiers such as public, private, final, static have the same meaning as in the case of Java declarations. We use the BML syntax to denote the logical connectives i.e., && means the logical conjunction, || the logical alternative, and ! the logical negation. The logical implication is denoted as \(\Rightarrow\). We also use the BML syntax to access and initialize elements of arrays and to denote types of variables. We also use here the general quantifier. The syntax of the quantifier expression is as follows:

\[ (\forall x \in X) \text{declaration} \text{bound on the quantification} \text{formula} \]

where the \text{variable declaration has the same form as a variable declaration in Java and introduces the quantified variables}. The \text{bound on the quantification is a formula the goal of which is to restrict potentially infinite domain of the quantification to be finite and it can be any boolean expression}. At last, the \text{actual formula} is the \text{formula we are interested in}. A \text{bound on the quantification} \text{and an actual formula} are understood as the implication \(B \Rightarrow A\).

3.2 Data to translate

The annotations we need are of two kinds. First of all, a reliable description of the security requirements and the program structure must be provided at the side of BML i.e. the security levels of fields, method signatures, cdr structure etc. must be represented in the form of ghost variables. Therefore, the first group of the translated data consists of:

- a table \(\Gamma\) of method signatures,
- a global policy \(\pi\) that provides security levels of fields,
- a cdr structure \((\text{region}_\text{a}, \text{jan}_\text{a})\)\(^7\),
- functions classAnalysis, excAnalysis, nblocs, nbargs, Handler.

The idea of our translation is that we check by means of the BML formulae that a derivation in the information-flow type system is correct. Therefore, we must translate the data on which the type system operates. To this end we need to transform the functions mentioned in Def. 1 i.e. for each policy signature of a method: a security environment \(\pi\) : \(PP \rightarrow S\) and a function \(\eta : PP \rightarrow S\).

Additionally, we use certain static data which is not defined explicitly in terms of ghost variables, but is included in the definitions below. The values of this kind are:

- \(\text{maxNbArg}\) the number of all exception types in \(P\).
- \(\text{maxS}\) the maximal security level used to type-check \(P\); the level 0 \(\notin S\) will be used to mark the undefined value. Let \(S_0 = S \cup \{0\}\).
- \(\text{maxNbArg}\) the maximal number of arguments of all methods in \(P\).

---

\(^7\) We actually do not provide the definition for \(\text{jan}_\text{a}\) as the function does not occur in the typing rules in [4].
\[ P_m[i] = \text{ifeq } j \ \forall j' \in \text{region}(i, 0), k \leq \text{se}(j') \]  
\[ \Gamma, \text{ft, region, se, } k_p \text{, } h \text{, } i \vdash k : st \implies \text{lift}_i(st) \]

Intuitively, the function \( \text{se} \) gives for each program point \( i \) a security level \( k \) such that the instruction at \( i \) cannot store to locations of a level lower than \( k \); the function \( \text{ft} \) associates with each program point a stack of security levels such that the operands on the actual stack cannot be at level higher than the one indicated by \( \text{st} \), at last \( \text{sgn} \) is the currently used security signature for the analyzed method.

Let us analyze the rule on Figure 1. The assertion under the line assumes that given are: a table of method signatures \( \Gamma \), security levels of fields \( \text{ft} \), a region function, a security environment function \( \text{se} \), a signature of the current method on \( k_p \), a region \( \text{region} \), and a program point \( i \). It asserts safety for the cases when the normal (non-exceptional) step is taken by the program. A safe step, here, must transform in the abstract world the stack \( h : st \) to a stack \( \text{lift}_i(st) \). This rule allows to assert it provided that two requirements are fulfilled. The first one \( P_m[i] = \text{ifeq } j \) requires that the instruction at point \( i \) is \text{ifeq}. The second one, \( \forall j' \in \text{region}(i, 0), k \leq \text{se}(j') \), describes the requirement on the code executed after the branch. The level \( k \) is the security level of the value read by \text{ifeq}. All program points in \( \text{region}(i) \) are points executing under the guard of \( i \). Since security environment \( \text{se} \) is meant to be the upper bound of all the guards under which the program point execute, it is natural that \( k \leq \text{se}(j') \) for all \( j' \in \text{region}(i) \). The stack on the right-hand side of \( \implies \) is lifted; \( \text{lift}_i \) is the point-wise extension to stack types of \( \text{lift} \). The lifting operation prevents illicit flows through the operand stack; the stack shows what happened before, for example what was the value at \text{ifeq}, and lifting prevents leaking of this information.

Note that \( \text{st} \) and \( \text{se} \) are chosen for particular signature \( \text{sgn} \). This signature, in turn, comes from \( \text{Policies}(m) \) and for each security level \( s \in S \) we have a single signature. In this light we may consider \( \text{st} \) and \( \text{se} \) as collections indexed by elements of \( S \).

The main theorem of \( \Box \) states that typable programs are non-interfering.

### 3 Translation from the information flow system

In order to express in the bytecode program logic that there is no unwanted information flow in the program \( P \) we need to add some formula annotations to \( P \) and to extend the method specification tables for \( P \) with the formulae which encode the conditions from Def. 1. Both can be done separately for each method. The translation we present here uses extensively BML and in particular ghost fields of the formalism.

#### 3.1 Summary of BML

The translation of the information flow system into the bytecode program logic is done with the use of the BML syntax. The BML formulae can be translated to the actual bytecode program logic with the use of the translation in [12]. Here is a brief
Lua, is somewhat similar to Python according to its characteristics, a multi-paradigm programming language that allows functional, imperative and object-oriented (based on prototypes) programming techniques. It includes a lightweight—it has just 38 instructions—and fast execution environment based on a register architecture [16].

It is worth noting that our results are not restricted to those programming languages only. Actually, we conjecture that this is true for the interpreters of programming languages with similar characteristics, i.e., for the Python case this also includes Perl [21], and Ruby [14].

### 2.1 Categorization based on the comparative addition example

Our classification scheme requires the assessment of the abstraction level of several virtual machines interpreting different languages. In order to do so, we take a representative bytecode instruction present in all our candidates and analyze their implementations. This enables us to show important differences in bytecode implementation, and in consequence allows us to classify them accordingly.

The representative instruction we use for demonstration is integer addition, e.g. for Java we take a look at `IADD`, for OCaml we show `ADDINT`, for Python `BINARY_ADD`, and for Lua we inspect `OP_ADD`. With the notable exception of Lua, all our candidates use a stack architecture, i.e., they need to pop their operands off the corresponding stack, and push their result onto it before continuing execution. In Lua’s register architecture, operand-registers and result-registers are encoded in the corresponding stack, and push their result onto it before continuing execution.

In our classification architecture, operand-registers and result-registers are encoded in the instruction.

We have highlighted the relevant implementation points by using a bold font, and use arrows for additional clarity.

```plaintext
case SWINSTRUCTION_IADD:
    
    jnt value1 = stack[stack_size - 2].jint;
    jnt value2 = stack[stack_size - 1].jint;
    stack[stack_size - 1].jint= value1 + value2;
    break;

Fig. 2. Implementation of Java’s integer addition operation, IADD in Sable VM v1.13.

```Instruct(ADDINT):

```plaintext
accu = (value)((intnat) accu *(intnat) *sp++ - 1);
Next;
```

Fig. 3. Implementation of integer addition in OCaml v3.11.0.

Figures 2 and 3 share an interesting characteristic. They do not implement addition on the virtual machine level, but express the bytecode addition by leveraging the addition used by the compiler, i.e., expose the addition of the native than one successor.

We assume that a bytecode program P comes equipped with a control dependence regions structure cdr which consists of a pair of partial functions (region, jun). The role of the functions is to arrange the program into compact parts for which the analysis of program invariants can be conducted separately. The region function describes the internal parts of these regions while the jun function the connections between them.

The types of the functions are the following:

\[
\text{region}_m : PP \times (\{0\} + C) \rightarrow P(PP) \quad \text{jun}_m : PP \times (\{0\} + C) \rightarrow PP
\]

The functions can be axiomatized by the SOAP (Safe Over APproximation) properties [5, Sect. 4] ensuring that the control dependence regions structure correctly describe information flow in a program P.

**Typable programs** To check that a program is non-interferent one may use a type system presented in [3]. In this type system, every method is checked against its signatures separately. The type system is parameterized by:

- a table Γ of method signatures,
- a global policy ft that provides security levels of fields,
- a cdr structure (region, jun) for every method m.

We assume also that the functions below are given and that they are correct:

1. classAnalysis which for a program point returns the set of exception classes of exceptions that may be thrown at the program point.
2. excAnalysis which for a method name N(m) returns the set of exception classes that are possibly thrown by m.
3. sblocs which for a method name N(m) returns the number of its local variables.
4. sbargs which for a method name N(m) returns the number of its arguments.
5. Handler which for a given point i in the method m and an exception e returns the point where the handler of the exception starts.

**Definition 1 (typable programs and methods)** Method m is typable with respect to ft, Γ, region, and a signature sgn if there exists a security environment se : PP → S and a function st : PP → S such that

\[
\text{st}(0) = \epsilon \quad \text{and for all } i, j \in PP, \epsilon \in \{0\} \cup C:
\]

1. if \(i \prec \epsilon \) then there exists \(s \in S\) such that \(\Gamma, ft, region, se, sgn, i \vdash \text{st}(i) \Rightarrow (s \in S, \epsilon)\),
2. if \(i \prec \epsilon \) then \(\Gamma, ft, region, se, sgn, i \vdash \text{st}(i) \Rightarrow \) where \(\epsilon\) denotes the point-wise partial order on the type stack with respect to the partial order taken on security levels. An example of a rule to derive \(\ldots \vdash \epsilon \Rightarrow \left\{\epsilon\right\} \) and for ifeq is given in Figure 3.

A program P is typable with the policy (h_{\text{in}}, ft, Γ) and cdr satisfying SOAP if every method m from P is typable with respect to ft, Γ, region, and all signatures in Policies(m).

7 We limit our exposition to the ifeq instruction due to the space restrictions.
is unable to observe the termination of the programs.

Formally, a security policy is expressed in terms of a finite partial order \( (S, \leq) \). This order allows to describe the capabilities of the attacker and the program to be analysed:

- A security level \( k_{\text{obs}} \) determines the observational capabilities of the attacker (she can observe fields, local variables and return values the level of which is less or equal than \( k_{\text{obs}} \)).
- A policy function \( f \) assigns to each field its security level. This allows us to express the non-interference property we are interested in:
- A policy function \( \Gamma \) that associates to each method identifier \( N(m) \) and security level \( k \in S \) a security signature \( \Gamma_N[m,k] \). This function gives the security policy of the method \( m \) called on an object of the level \( k \). The set of security signatures for a method \( m \) is defined as \( \text{Policies}(m) = \{ \Gamma_N[m,k] | k \in S \} \). The security signature has the shape \( P_p \leq \Gamma_p \sqcup \Gamma_e \):
  - The vector \( \Gamma_p \) describes the security levels appropriate for the local variables of the method (in particular it assigns also the levels to the input parameters), \( k_p[k] \) is the upper bound on the security level of an object that calls the method.
  - The value \( \Gamma_e \) describes the lower bound in the security levels of the heap operations performed by the method.
  - The vector \( \Gamma_k \) describes the security levels for the method results (both normal and exceptional ones); it is a list of the form \( \{ e_1 : \Gamma_{k_1}, e_2 : \Gamma_{k_2}, \ldots, e_n : \Gamma_{k_n} \} \), where \( k_k \) is the security level of the return value and \( e_i \) is the class of an exception that might be thrown in the method and \( k_k \) is the upper bound on the security level of the exception. We use the notation \( k_{\text{obs}}(n) \) and \( k_{\text{obs}}(e) \) for \( k_k \) and \( k_e \).

Non-interference

The non-interference property is articulated by a safety definition in [45]. Informally, a program is non-interferent if all its methods are safe; a method is safe if no terminating run of the method with inputs that cannot be distinguished by the attacker and if the method cannot modify the heap in a way that is observable by an attacker.

Non-structured programs

The bytecode programs organise the control flow by means of jump instructions. In order to reason on the information flow of such programs an additional structural information is needed. As we translate typings in an information flow system [5], we need the same descriptions of the bytecode program structure.

We use a binary successor relation \( \rightarrow \subseteq PP \times PP \) defined on the program points. This relation is parameterised by a tag \( r \) since instructions may have several successors as they may execute normally (the tag is 0) or may trigger exceptions (the tag is the class of the exception). Intuitively, \( j \) is a successor of \( i \) \( (j \rightarrow i) \) if performing one step execution from a state whose program point is \( i \) may lead to a state whose program point is \( j \). We write \( i \rightarrow j \) when \( i \rightarrow j \) is undefined for \( i \) or if \( i \) corresponds to a return instruction (or \( i \rightarrow j \) if \( i \) corresponds to an instruction that may throw an exception that is not handled locally). Note that an instruction may have more

---

3 In [5] a more precise notation \( \Gamma_N[m,k] \) is used.
not restricted by native machine boundaries in any way. As a direct consequence, the original Python add instruction cannot be directly mapped onto one native machine instruction in the interpreter. This constitutes our second class, namely high abstraction level virtual machines.

```
OP_ADD:
  if (ttisnumber(rb) && ttisnumber(rc)) {
    lua_Number nb = nvalue(rb), nc = nvalue(nc);
    setvalue(ra, (nb + nc));
  } else {
    L->savedpc = pc;
    Arith(L, ra, rb, rc, TH_ADD);
  }
  base = L->base;
```

Fig. 6. Implementation of integer addition in Lua 5.1.4.

Our classes are by no means completely separated and disjoint, since interpreters can be members of both classes, having some subset of instructions belong into one set, and a separate subset of instructions to the other. This is exemplified in Lua (cf. Figure 6) in which addition has characteristics of both classes.

In Lua, if operand types are numeric it delegates the actual addition implementation to the interpreter’s add instruction, and how the actual add is realized using a single machine add instruction.

2.2 Comparison of Low and High Abstraction Level Virtual Machines

The previous section introduces our two classes of interpreters, namely:

- Low abstraction level, where operation implementation can be directly translated to a few native machine instructions. Figure 7(a) shows the implementation of the interpreter’s add instruction, and how the actual add is realized using a single machine add instruction.

- High abstraction level, where operation implementation requires significantly more native machine instructions than for low abstraction level. Analogous to Figure 7(a), Figure 7(b) shows the relative impact of implementing a complex add. Frequent characteristics for high abstraction level are:

2 Preliminaries

In this section we fix the notation used throughout the paper. We, generally, follow the papers [4, 12]. We also present informal description of some notions which are not directly used in the translation, but are essential for understanding the principles of the construction.

Basic notation. We use the expression dom(f) to denote the domain of the (partial) function f. Similarly, rng(f) is the image of f. We write f : A → B when f is a partial function from A to B. We assume that the set A is P(A). We write k to denote a vector of values. The notation [k] expresses the length of the vector and k(0), . . . , k(|k| − 1) are the elements of the vector. The set of finite sequences over a set A is A*. The separation of the identities for the method and its name serves to model the inheritance.

Java bytecode programs and specifications. A Java bytecode program P is a set of classes with one single out method main. A class C is a set of fields and methods. Each field f has a name fs and a type ft. Similarly, a method m has a name N(m), a signature Tm and a body Bm. We assume that the method names are unique within a single program (possibly due to the standard Java prefixing with an object or class name). A method body is a sequence of bytecode instructions. The instructions are indexed by program points. For each method m we distinguish the set of all program points in the method Pmprm (we omit the subscript m when it is clear from the context).

An annotated program P has additionally (among others) for each class C a list methods C of model and ghost fields (i.e. fields which can occur only in specifications), and a method specification table Mc. The bytecode program logic we employ here [7] makes use of the method specification table Mc(m) associated with each method m. This table consists of:

- a method specification Sm = (Rm, Tm, Pm) where Rm is the precondition of the method, Tm is the postcondition of the method, and Pm is the method invariant which holds in each accessible state of the method;
- a local specification table Gm which assigns to each label in the method body Rm an additional assumption that may be used in the proof of the program verification clause associated with the label (this corresponds to the BML assume annotations);
- a local annotation table Qm which assigns to labels in Rm further assertions (this corresponds to the BML assert annotations);
- a local instruction table ltm which assigns to each label l in the method body Rm a sequence of bytecode instructions that operate on ghost variables which is supposed to be executed before the instruction at the label l and the respective specification Qm(l) (this corresponds to the BML set annotations).

Security policy. We use here the security policy framework from [5]. It is based on assumption that the attacker can observe the input/output of methods only. This, however, is extended to the values of fields and heaps as otherwise it is difficult to guarantee the non-interference property. We also assume that the attacker...
on the design-by-contract principles and is derived from a Java specification language called Java Modeling Language [18,19,20] which has wide tool support [21].

We consider here the non-interference property of bytecode programs. Many type systems guarantee the non-interference have been proposed for while programs (e.g., [16,17,21]) as well as for other formal languages (e.g., [14,15]). The starting point of our work is the information-flow type system for Java bytecode [4,7] that guarantees the non-interference for sequential bytecode programs with objects, methods and exceptions. The main contribution of the current paper is a translation of the type system into the bytecode program logic developed within the MOBIUS project [7,12] such that the correctness of the typing is equivalent to the verifiability of bytecode program logic formulae, Here, the soundness of the type system guarantees the non-interference property of veriable programs.

It is worth pointing out that the translation has a few desirable features. First of all, once the translation is in place it is possible to use a toolset based on logical methods rather than typed ones. This allows to incorporate the guarantees of the type system into a foundational proof-carrying code (PCC, [2]) framework and use the non-interference property together with other properties originally formulated and expressed in the foundational fashion 4. Moreover, the wide selection of JML-based verification tools and methods [10] is a solid basis to aim for a platform of foundational PCC certificates for Java bytecode. Another desirable feature of this method is the fact that the resulting model of the non-interference is more flexible than the one based on typing. This is important whenever the non-interference property must be weakened, for example when declassification is needed (in particular when the code encrypts confidential data). The translation we provide is designed so that it is relatively straightforward to adjust it to various declassification needs.

The Horn-like logic available in JML is in fact only first order logic with very weak inventory of relations which allow to compare heaps at different points of program. In particular, it is impossible to express there the agreement operator by Amtorf et al [1]. Moreover, JML also does not contain any features of dynamic or algorithmic logic so it is impossible to express the non-interference property by relating the heaps after two different program runs as in [6] or [13]. Therefore, we decided to model the type system derivations in JML and base the safety of the program on the type system soundness. Still another limitation of the approach based on JML is that the self-composition [5] cannot be expressed here (although it is available in MOBIRS logic).

The paper is structured as follows. In Sect. 2, we fix the notation and present the basic notions which are used in the paper. Sect. 3 provides an exposition of the translation of the type based system to the logic based one. This translation is supplemented by a theorem that the resulting specifications guarantee the non-interference property in Sect. 4. The formal development is concluded by a proof that the non-interference property holds even when the bytecode program is extended with other specifications. This is presented in Sect. 4.1. At last we present the final remarks in Sect. 5 where we sketch the way the declassification can be introduced.

![Fig. 7. Illustration of Virtual Machine Abstraction Levels. Important are the different ratios of bytecode instructions to application instructions.](image)

(a) Low Abstraction Level Virtual Machine
(b) High Abstraction Level Virtual Machine

- **Ad-Hoc Polymorphism**: a) either polymorphic operations are selected for concrete types of operand types, e.g., in Python, or b) operand-type coercion into compatible types for a given operation implementation, e.g. in Lua or Perl.
- **Complete Operation Implementation**: In Python’s case, this complexity directly maps to unbounded integer range mathematics for numeric operations, or full unicide support at the interpreter level.

3 Optimizations for Low Abstraction Level Interpreters

The well known techniques for interpreter optimization focus on reduction of instruction dispatch cost. As shown in Figure 7(a) virtual machines with low abstraction level are particularly well suited for those techniques, since dispatch often is their most expensive operation. Threaded code [5] reduces the instructions necessary for branching to the next bytecode implementation. The regular switch dispatch technique requires 9-10 instructions, whereas e.g. direct threaded code needs only 3-4 instructions for dispatch [6], with only one indirect branch. Superinstructions [9] substitute frequent blocks of bytecodes into a separate bytecode, i.e., they eliminate the instruction-dispatch costs between the first and last element of the replaced block.

Recent advances in register based virtual machines [20], however, suggest a complete architectural switch from a stack-based interpreter architecture to a register based model. This model decreases instruction dispatches by eliminating a large number of stack manipulation operations, i.e., the frequent LOAD/STORE operations that surround the actual operation. The paper reports that 47% of Java bytecode instructions could be removed, at the expense of growing code size of about 25%.

Table 2 shows a list of achievable speedups for low abstraction level interpreters. For high abstraction level virtual machines, these speedups are not nearly as high. Vitale and Abdelrahman [22] actually report that applying their optimization technique to Tcl has negative performance impacts on some of their benchmarks, because of instruction cache misses due to complex operation implementation leading to excessive code growth—the main characteristic we use to identify high abstraction level virtual machines.

This, however, does not mean that these techniques are irrelevant for virtual machines with a high abstraction level. Actually, quite the opposite is true: once

---

4 The translation from this paper does not reduce the trusted logical base to the one of the foundational PCC. This is not the case with the non-interference property expressed in the foundational logic, e.g., the property expressed in [8] for while programs.
4 Optimizations for High Abstraction Level Interpreters

Figure 7(b) shows that many machine instructions are necessary for realizing the high abstraction level of an interpreter instruction. Therefore we are interested in cutting down the costs here, since they provide the greatest speedup potential. In this situation it makes sense to provide a reminder as to what characteristics constitute our classification into the class of high abstraction level. As already mentioned earlier in the addition example, there are two answers to that question:

a) ad-hoc polymorphism
b) complex operation implementation (unbounded range mathematics)

Hence there are two issues to deal with. In the first case (a), a look at the history of programming languages provides valuable insights. We are trying to find programming languages with similar characteristics like Python’s but having more efficient execution environments. Smalltalk fits the bill, and specifically SELF is a prominent derivative which offered an efficient execution engine back in the early 90s. Among the various optimizations in SELF, specifically type feedback in combination with inline caching seems particularly matching our first problem. Holde and Ungar [12] report performance speedups by a factor of 1.7 using type feedback, and give advice that languages having generic operators are ideally suited for optimization with type feedback. Application of type feedback requires that for a pair of operand-types the target of the actually selected operation implementation is cached, such that consecutive calls can directly jump there, when operand-types

---

### Table 2

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Speedup Factor</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threaded Code (compared to switch dispatch interpreter)</td>
<td>up to 2.02</td>
<td>[8]</td>
</tr>
<tr>
<td>Superinstructions (compared to threaded code interpreter)</td>
<td>up to 2.45</td>
<td>[7]</td>
</tr>
<tr>
<td>Replication + Superinstructions (compared to threaded code interpreter)</td>
<td>up to 3.17</td>
<td>[7]</td>
</tr>
<tr>
<td>Register vs. Stack Architecture (both using switch dispatch)</td>
<td>1.323 avg</td>
<td>[20]</td>
</tr>
<tr>
<td>Register vs. Stack Architecture (both using threaded code)</td>
<td>1.265 avg</td>
<td>[20]</td>
</tr>
</tbody>
</table>

References of reported speedup factors for several techniques.
match the cached pair (cf. Figure 8).

Fig. 8. Type feedback for Python’s \texttt{BINARY ADD} instruction. The gray-colored part is the system default look-up routine from Figure 5. The dashed arrows represent querying the type feedback cache (upper bidirectional arrow), and updating the cache with new target addresses after running through the system look-up routine (lower unidirectional arrows).

In the second case (b), the same technique can be applied. This requires that the actual operation implementation would be sub-structured to the following steps:

(i) Try to use the fastest possible native machine method

(ii) If it fails/overflows, apply unbounded range software algorithm

By encoding this information into separate types—e.g. int, long, arbitrary—, the type feedback infrastructure of our first problem (a) can be reused. In such a case, a positive check against machine boundaries and overflow errors, calling downwards the chain of most-general implementations and updating the cache for subsequent calls is necessary.

Aside from these optimization techniques, we want to mention a subtle issue that comes up when comparing high abstraction level instructions with low abstraction level instructions. When we compare the addition example of Java and Python, we find that JVM’s integer addition bytecodes, \texttt{IADD} and \texttt{LADD}, are bound by a maximum range of representable numbers—32-bit for integers, and 64-bit for long integers respectively—whereas Python’s \texttt{BINARY ADD} implementation is not. Since the JVM does not offer unbounded range mathematics at the virtual machine level, it is necessary to leverage library functionality—in our case \texttt{java.math.BigInteger}—in order to have a 1 : 1 correspondence between the integer addition of both languages. In Java’s case a call to \texttt{IADD}, or \texttt{LADD} for that matter, would be substituted by an invocation of a software algorithm for unbounded range mathematics of \texttt{java.math.BigInteger}—probably similar to the one implemented for Python’s \texttt{BINARY ADD}, or \texttt{long add} respectively. This implies that even though a similar algorithm might be used, their difference in implementation level is significant: in Java we need a library, which generates multiple bytecodes for implementation of the unbounded range addition, whereas the Python compiler still emits just a single \texttt{BINARY ADD} instruction.

Consequently, a high abstraction level sometimes can be considered as an optimization technique itself, since it can save a considerable amount of emitted low
abstraction level instructions—we could probably say that this is a derivative, or special case, of the superinstruction optimization technique [9]. In conclusion, this example also illuminates that the instruction set architecture is also of significant importance for the performance of virtual machines—probably we can also reuse the analogy of hardware machine instruction set architecture, by recognizing the terms RISC and CISC in context with our classification, e.g. low and high abstraction level interpreters.

5 Related Work

Romer et al. [19] provide an analysis for interpreter systems. Their objective was to find out whether interpreters would benefit from hardware support. However, they conclude that interpreters share no similarities, and therefore hardware support was not meaningful. Among other measurements, they collected average native instructions per bytecode (Section 3 of their paper). This is a sort of a black-box view on our classification scheme based on comparable source code examples from bytecode implementations. Finally, there is no link to optimization techniques, too.

Contrary to Romer et al. [19], Ertl and Gregg [8] found that at least within the subset of efficient interpreters, hardware support in the form of branch target buffers would significantly improve performance for the indirect branch costs incurred in operation dispatch. Their in-depth analysis by means of simulation of a simple MIPS CPU found that the indirect branching behavior of interpreters is a major cause for slowdowns. Another important result of Ertl and Gregg is that a Prolog implementation, the Warren Abstract Machine based YAP [4], is very efficient, too. This implies that there is no inmanent performance penalty associated with dynamically typed programming languages. Their class of efficient interpreters maps perfectly well to our category of low abstraction level virtual machines.

Adding to their set of efficient interpreters, they also provide results for the interpreters of Perl, and Xlisp—both of which achieve results that do not fit within the picture of efficient interpreters. This is where we introduce the concept of high abstraction level interpreters, and how it correlates to optimization techniques. Interestingly, for Xlisp Ertl and Gregg mention the following: “We examined the Xlisp code and found that most dynamically executed indirect branches do not choose the next operation to execute, but are switches over the type tags on objects, Most objects are the same type, so the switches are quite predictable.” This directly translates to our situation with high abstraction level interpreters (Section 4).

In his dissertation, Hölzl also notes that a problem for an efficient SELF interpreter would be the abstract bytecode encoding of SELF [11], with a point in case on the send bytecode, which is reused for several different things. Interestingly, Hölzel observes, that instruction set architecture plays a very important role for the virtual machine, and conjectures that a carefully chosen bytecode instruction set could very well rival his results with a native code compiler.


6 Conclusions

We introduced the classes of high and low abstraction levels for interpreters, and categorized some interpreted systems into their corresponding classes. Using them, we subjected various known optimization techniques for their relative optimization potential. Techniques that achieve very good speedups on low abstraction level interpreters do not achieve the same results for high abstraction level virtual machines. The reason for this is that the ratio of native instructions needed for operation execution vs. the native instructions needed for dispatch, and therefore their relative costs changes. In low abstraction level interpreters the ratio usually is 1:n, i.e., many operations can be implemented using just one machine instruction, but dispatch requires n instructions, which varies according to the dispatch technique applied, and is costly because of its branching behavior. Quite contrary for high abstraction level interpreters: here operation execution usually consumes much more native instructions than dispatch does, which lessens the implied dispatch penalties.

Our classes are not mutually exclusive, an interpreter can have both, low and high abstraction level instructions. For our classes of high abstraction level interpreters, exemplified by Python and Lua— but conjectured to be true for Perl and Ruby, too—type feedback looks particularly promising. When faced with other programming languages but the same situation, i.e., a discrepancy in expected and reported speedups for low abstraction level techniques, our mileage may vary. In such a situation, only detailed analysis of an interpreter’s execution profile can tell us where most time is spent and which techniques are most promising with regard to optimization potential.

In closing, we want to mention that our objectives are to demonstrate the relative optimization potential for different abstraction levels between an interpreter’s virtual machine instruction set and the native machine it runs on.

Acknowledgments

I want to thank Jens Knoop and Anton Ertl for their feedback and comments on earlier versions of this paper, which helped to clarify issues and to improve the presentation. I would also like to thank the anonymous reviewers for their helpful comments and remarks.

References


Fig. 1. The termination analyses of some programs. Times in seconds. The second line (precisions), for each program, reports the number of methods preceded to terminate. In the header, we refer to the subsection where the simplification is described.

References

This proposition justifies the removal of a uniform variable from a strongly-connected component.

**Proposition 4.17 (Correctness of a uniform variable removal)** Let a variable be uniform in a strongly-connected component Comp. If an entry diverges in Comp\(^{-v}\), then it diverges Comp.

**Proof.** Let \( x \in \mathbb{Z} \) as in Definition 4.14. From an infinite resolution in Comp\(^{-v}\), we can construct an infinite resolution in Comp by simply inserting \( x \) into each call at the position of variable \( v \).

If we apply this simplification to the CLP program obtained at the end of Subsection 4.5, the number of clauses remains 6 but there are less arguments in predicates. The time needed to prove all the entries terminating goes down from 1.02 to 0.67 seconds (including the time to apply all the simplifications).

### 5 Experiments

Figure 3 reports the results of our termination analysis and the effects of our simplifications on the time needed to build a proof of termination for the entries of the program. **Ackermann** is an implementation of the traditional Ackermann function. **Bubblesort** is an implementation of the bubblesort algorithm on arrays. **NQueens** is a program that solves the n-queens problem by using a library for binary decision diagrams, included in the analysis. **Lex** is a lexical analyzers generator. **Kitten** is a didactic compiler for simple object-oriented programs. Our experiments have been performed on a Linux machine based on a 64 bits dual core AMD Opteron processor 280 running at 2.4Ghz, with 2 gigabytes of RAM and 1 megalobyte of cache, by using Sun Java Development Kit version 1.5 and SICStus Prolog version 3.12.8. For each program, we report the number of methods (without the Java libraries) and the time for building a proof of termination with the original, unoptimized technique of [17] and with the successive application of more and more simplifications, described in this paper (the time for the simplifications is included). The header of each column reports the subsection where the simplification is described. The original technique failed to conclude the analysis after 15 minutes for **NQueens**, **Lex** and **Kitten**. In general, more simplifications means better efficiency. This relation is not always true. For instance, building a proof of termination for **Lex** takes 228.51 seconds if only the simplification of Subsection 4.1 is applied. If also the simplification of Subsection 4.2 is applied, this time increases to 335.85. We explain this behaviour with the fact that simplifications have a cost. Moreover, when the program is too complex, **BinTerm** uses timeouts, which makes the construction of the proof faster. However, the precision of the proof decreases with the number of timeouts. Hence, below each program, we report the number of methods proved to terminate. This number increases with the number of simplifications applied to the CLP program, since less timeouts are triggered.
Towards an XML-based Bytecode Level Transformation Framework

Arno Puder1 and Jessica Lee2

San Francisco State University
Computer Science Department
Joff Reitblatt, Assistant
San Francisco, CA 94132

Abstract
Virtual machines (VMs) facilitate the deployment of applications in heterogeneous environments. Popular VMs such as Sun Microsystems’s Java VM or Microsoft’s Common Language Runtime (CLR) use a stack-based machine architecture to execute bytecode instructions. The focus of this paper is to explore the ability of XML to serve as a generic framework to represent bytecode instructions from different VMs. With an XML representation, existing technologies such as XSL style sheets and XPath expressions can be used to provide various transformations of bytecode instructions. We demonstrate the flexibility and power of this approach by showing examples of cross-compilation for various CLR bytecode instructions to the JVM as well as API mappings of the underlying runtime libraries.

1 Introduction
Virtual machines are abstract computing machines that offer a homogeneous computing platform in a heterogeneous environment. Instead of compiling source code to machine language, an intermediate language called ‘bytecode’ is produced. It cannot directly run on a physical machine and requires a virtual machine that loads and executes the application. While often confronted with the argument of being slower in execution, they offer a number of advantages over machine compiled languages. Specifically, they ease the development and deployment of applications in heterogeneous environments, without the need to recompile the application for a specific platform. Two virtual machines that are widely used today are the Java Virtual Machine (JVM) from Sun Microsystems [9] and Microsoft’s Common Language Runtime (CLR) as part of their .NET framework [2]. Implementations can be found on devices like cell phones, personal computers, or chip-cards. Both virtual machines rely on a stack-based execution model, but the CLR features a wider range of data types and bytecode instructions than the JVM.

1.6 Removing uniform variables
Even if an isolated variable is neither left-open nor right-open, it can still be removed when there is a fixed value that can be put in that variable throughout an infinite resolution. Such a variable is called uniform.

Definition 4.14 [Uniform variable] An isolated variable \( v \) is uniform in a strongly-connected component Comp if there is \( x \in Z \) such that, for every \( p(x) \vdash q(o) \) in Comp, the valuation \( \{v = x, o = x\} \) is a solution of \( c^v \) (note that \( c^v \) may contain more than one constraint).

Example 4.15 The program obtained at the end of Subsection 4.5 contains the component:

```prolog
block3853(1L0,1L1,1L2) :- 1L2 = 0L2 = 1L1 = 0L1 = 0L0 = 0L0 = 0, 1L1 = 1L2 = 1, block3853(0L0,0L1,0L2).
block3853(1L0,1L1,1L2) :- 1L0 = 0L2 = 1, 1L1 = 0L1 = 0L0 = 1, 1L1 = 1L2 = 2, entry3849(0L0,0L1,0L2).
entry3849(1L0,1L1,1L2) :- 1L2 = 0L2 = 0L1 = 0L1 = 0L0 = 0L0 = 0, 1L0 = 1, block3853(0L0,0L1,0L2).
```

By taking \( x = 1 \), we conclude that \( 1L0 \) is uniform.

Example 4.16 Uniform variables and left- or right-open variables are different concepts. For instance, variable \( 1L0 \) is uniform in the component of Example 4.15 but it is not left-open or right-open. Conversely, variable \( 1L1 \) is also uniform in the component

\[
\text{entry3849}(ix,iy) \vdash iy \geq 0, ax = ix, ix \geq 3, ox = iy - 1, p(oz,oy),
\]

\[
p(ox,ky) \vdash ax = ix, ix \leq 0, oy = iy, entry3849(ix,oy).
\]
Programs to manipulate virtual machine bytecode are becoming more prevalent. BAT2XML uses XML to represent Java bytecode in order to take advantage of supporting XML technologies for processing and manipulation. One of the features of BAT2XML allows for the easy injection and extraction of Java bytecode. This means that the bytecode can be analyzed and then optimized. BAT2XML is only able to represent Java bytecode, however, and has no provisions for CLR bytecode.

Aspect-Oriented Programming (AOP, [7]) provides another example of the need to manipulate bytecode. AOP is concerned with separating the basic business logic of a system from what it deems to be crosscutting concerns such as logging and instrumentation. These concerns are shared by modules throughout a system. Rather than being added at the source code level, they are separated out into aspects and added in at compile time by an aspect weaver. This “weaving” is done on a bytecode level.

Cross-compilation is yet another example of a situation where bytecode is directly processed. It would be of obvious economic interest to be able to map bytecode instructions from one VM to the other. By doing so, developers for one VM could deploy their applications on the other VM. The IKVM project [4] offers support to execute JVM applications on a CLR platform. The JaCIL project [5] cross-compiles .NET executables, the .NET framework offers a similar library based on a reflection model. Rather than being added at the source code level, they are separated out into "authorization". These concerns are shared by modules throughout a system. Ultimately, this paper is a showcase for the power of XML technologies and supporting XML technologies for processing and manipulating. One of the features of BAT2XML allows for the easy injection and extraction of Java bytecode. This means that the bytecode can be analyzed and then optimized. BAT2XML is only able to represent Java bytecode, however, and has no provisions for CLR bytecode.

The approach taken in this paper is to make extensive use of XML technologies in order to provide a framework for generic bytecode manipulation. We use XML in order to provide a representation of bytecode that is a superset of stack-based machine languages. This allows for the easy manipulation of bytecode, and we demonstrate this with a cross-compilation example that uses XSL stylesheets [10] for translating CLR bytecode instructions to the JVM. XSL stylesheets allow for a declarative methodology when performing this transformation. Ultimately, this paper is a showcase for the power of XML technologies and declarative programming for a non-trivial application. The outline of this paper is as follows: In Section 2 we present XMLVM, our XML-based representation of bytecode instructions. Section 3 gives an overview of the JVM and the CLR while focusing on their differences. Based on XMLVM, we show in Section 4 how to cross-compile bytecode instructions using XSL stylesheets. Section 5 presents conclusions and outlook for future work.

2 XMLVM

Cross-compilation requires access to the bytecode instructions for both the CLR and the JVM. From an engineering perspective, this can be accomplished by various libraries that allow the inspection and construction of executables. For Java class files, the Byte Code Engineering Library (BCEL, [1]) allows such manipulations. For CLR executables, the .NET framework offers a similar library based on a reflection API for low-level bytecode manipulations. It should be noted that BCEL can only be accessed using Java while the .NET API is only offered for languages supported by .NET. Currently, there is no single library that allows JVM and CLR bytecode manipulations from one high-level programming language (e.g., Java or C#). Any

Example 4.12 The program obtained at the end of Subsection 4.3 contains the component

```
entry3880(1L,1L1):-IL1 - OL1 = 0,OL1 >= 0,IL0 >= 2,IL0 - OL0 >= 1,
entry3880(0L0,0L1),
```

where variable L1 is both left- and right-open and can hence be removed obtaining the component

```
entry3880(1L0):-OL0 >= 0,IL0 >= 2,IL0 - OL0 >= 1,entry3880(0L0).
```

L1 would still be left-open if there were an extra constraint IL1 >= 3. It would not be left-open anymore if there were also an extra constraint OL1 >= 7.

Consider a resolution of pℓ(jℓ) in a strongly-connected component Comp where v is right-open. Let pℓ(jℓ) = cℓ(qℓ(oj)) be the clause used at the jth resolution step. If cℓ is in or then the jth step simply copies the value of v from pℓ to pℓ+1. Otherwise, the value of v in pℓ is not related to that in pℓ+1: any value satisfying the v-dedicated part cℓ of cℓ may be picked up for v in pℓ+1: such a value exists always due to the limited form of cℓ. This means that v does not contribute to the termination of the predicates in Comp and can hence be removed. This is formally proved below.

Proposition 4.13 (Correctness of left- or right-open variable removal) Let v be right- or left-open in a strongly-connected component Comp. If an entry diverges in Comp," then it diverges in Comp.

Proof. We only prove the case when v is right-open. The case when v is left-open is symmetrical. Let hence pℓ be a divergent entry in Comp. Then there is mℓ ∈ Z and an infinite resolution of pℓ(mℓ) in Comp," which we write as

```
d0 → cℓ1 → cℓ2 → ... → cℓn → d1:
```

where every clause pℓ(i) ⊕ cℓj(qℓ(oj)) ⊕ v used in each portion di for i ≥ 0, is obtained from a clause pℓ(i) ⊕ cℓj(qℓ(oj)) ⊕ v used in each portion dj for each . Assume that

```
dj = (pℓ(jℓ)(mℓ,jℓ) → ... → pℓ(jℓ)(mℓ,jℓ+1) → pℓ(jℓ+1)(mℓ,jℓ+1) → ... → pℓ(j′)(mℓ,j′))
```

with pℓ(jℓ) = pℓ and mℓ0 = mℓ0. Let pℓ(jℓ)(mℓ,j) = pℓ(jℓ)(mℓ,j) ⊕ v for each 0 ≤ j ≤ j. Since pℓ(jℓ)(mℓ,j) → pℓ(jℓ+1)(mℓ,j+1), there is pℓ(jℓ) ⊕ cℓj(qℓ(oj)) ⊕ v ∈ Comp such that pℓ(jℓ) ⊕ cℓj(qℓ(oj)) ⊕ v ∈ Comp and there is a solution θ of (cℓj(pℓ(jℓ) ⊕ v → pℓ(jℓ+1)(mℓ,j+1)) = (pℓ(jℓ+1)(mℓ,j+1) ⊕ v θ) where cℓj is v-independent, θ ∪ (v → xℓ, ov → xℓ) is a solution of (v → xℓ, ov → xℓ) and (θ ∪ (v → xℓ, ov → xℓ)(cℓj+1(pℓ+j+1(oj)) = (pℓ+j+1(oj) ⊕ v θ) ⊕ v[xℓ](cℓj+1(mℓ,j+1) ⊕ v θ)

Bytecode 2009 32 ETAPS 2009, York, UK

Bytecode 2009 113 ETAPS 2009, York, UK
since $c^{-1}$ is $v$-independent. Thus,

\[(p_{j+1}(o_j) \lor v) \land p_j(o_j) \lor v = p_{j+1}(n_{j+1}) \lor v\]

and we can build the following infinite resolution of $p_0(n_0) \lor v$ in $\text{Comp}^x$:

\[p_0(n_0) \lor v \rightarrow p_1(n_1) \lor v \rightarrow p_2(n_2) \lor v \rightarrow \cdots \rightarrow p_k(n_k) \lor v \rightarrow \cdots\]

so that $p_0$ diverges in $\text{Comp}^x$.

In general, $\text{Comp}$ is not termination-equivalent to $\text{Comp}^x$.

**Example 4.8** Consider the strongly-connected component

\[
\text{Comp} = \left\{ (entry1(ix, iy) : ix \geq 0, oy = ix, ox = iy, q(ix, oy)) \right\}
\]

The entry $entry1$ terminates in $\text{Comp}$ since the value of $x$ decreases in every two other steps and is bounded from below by $0$. By removing $x$ from $\text{Comp}$ we get

\[
\text{Comp}^x = \left\{ entry1(iy) : true, q(oy) \right\}
\]

Now $entry1$ does not terminate in $\text{Comp}^x$.

The following subsections identify special cases when removal of a variable maintains the termination-equivalence. A common condition is that the variable is isolated from other variables.

**Definition 4.9** A variable $v$ is isolated in a strongly-connected component $\text{Comp}$ if for every clause $q(ix) : c, q(oy) \in \text{Comp}$, we have $c = c' \land c^{-1}$.

**Example 4.10** Neither $x$ nor $y$ is isolated in the component $\text{Comp}$ of Example 4.8. Instead, both $x$ and $y$ are isolated in the component

\[
\text{Comp} = \left\{ entry1(ix, iy) : ix \geq 0, oy = ix, ox = iy - 1, q(ix, oy) \right\}
\]

\[
q(ix, iy) : ox = iy - 1, oy = iy, entry1(ix, oy) \right\}
\]

\[
4.5 \text{ Removing right-open/left-open variables}
\]

In this subsection we show a first example of a removal of variables for which the converse of Proposition 4.7 holds.

**Definition 4.11** (Right or left-open variable) An isolated variable $v$ in a strongly-connected component $\text{Comp}$ is right-open if, for every $q(ix) : c, q(oy) \in \text{Comp}$, we have that $c'$ is either true or $v = ov$, or $ov \geq \text{const}$, $ov = \text{const}$ or $ov \leq \text{const}$ (or equivalent), where $\text{const}$ is an integer constant. **Left-openness** is defined analogously by switching $ov$ with $iv$ in the definition of right-openness.

kind of cross-platform bytecode manipulations are therefore difficult to realize. For that reason we have chosen to create an abstraction for bytecode instructions using XML [11] and limit the use of platform-specific libraries only during reading and writing JVM and CLR executables.

We use XML to represent the contents of a Java class file as well as the contents of a CLR executable. Since the resulting XML document is structured according to the semantics of a program for a generalized virtual machine, we call it XMLVM. Another way to look at XMLVM is that it defines an assembly language for those virtual machines using XML as the syntax. The benefit of using XML is that it creates an abstraction in the sense that it hides the complexities of bytecode manipulation libraries. It thereby allows us to focus on the bytecode manipulations itself by exploiting powerful XML technologies such as XSL and XPath. In addition, the declarative nature of XSL will result in compact specifications for mapping CLR bytecode instructions to the JVM.

The following template shows the general structure of an XMLVM translation unit used for both JVM and CLR programs:

```
<xmlvm xmlns:clr="http://xmlvm.org/clr"

xmlns:jvm="http://xmlvm.org/jvm"

<signature>...</signature>

<code>...</code>
```

An XMLVM program consists of several classes, each contained in a separate translation unit. Each class can have one or more fields and methods. The attributes of the XML tags, which are not shown in the template above, give more details such as identifier names or modifiers. A method is defined through a signature and the actual implementation, denoted by the tags `<signature>` and `<code>` respectively.

We make use of XML namespaces to indicate the semantics of the various tags used in an XMLVM program. The tags shown in the template above are located in the default namespace and represent common features between the CLR and the JVM. Those features specific to the particular VM—such as different byte code instructions—are located in their respective namespace. Consider the following simple C# program that will serve as an example when discussing the mapping of numerical operators:

```csharp
using System;

class AddTest {
  public static void Main() {
    int a = 11;
    int b = 22;
    Console.WriteLine(a + b);
  }
}
```

Class `AddTest` has one public static method called `Main`. The method adds two integer values and prints the sum to the console. The following XML shows a
Any divergent resolution in $P_2$ is also a divergent resolution in $P$ since $P_2$ has less clauses than $P$. Hence it is enough to prove that for any divergent resolution in $P$ there is a divergent resolution in $P_2$. To that purpose, we prove that if $C_1 = (p_1 : q_1, q(o))$ is subsumed by $C_2 = (p_1 : c_2, q(n))$ then $p(m) \to^{c_1} q(n)$ implies $p(m) \to^{c_2} q(n)$ for any $p, q, m, n$ which entails that any derivation step using $C_1$ can be replicated by using $C_2$. Assume hence that $p(m) \to^{c_1} q(n)$. Then there is a solution $\theta$ of $c_1[x \mapsto m]$ such that $n = \theta(n)$. Then $c_1 \models c_2, \theta$ is also a solution of $c_2[x \mapsto m]$ and hence $p(m) \to^{c_2} q(n)$.

If we apply these simplifications to the CLP program obtained at the end of Subsection 4.2, the number of clauses decreases from 8 to 7 and the time needed to prove all the entries terminating goes down from 1.48 to 1.25 seconds (including the time to apply all the simplifications discussed up to now).

### 4.4 Removing variables

By removing an argument from the clauses of a CLP program, the time needed to build a termination proof of the program decreases, since less arguments means less variables in the data structure implementing the linear constraints and hence better efficiency. Moreover, by removing variables there are chances that distinct clauses get merged because one subsumes another (Subsection 4.3).

Let $c$ be a constraint and let $c^\ast = \exists_{(i \in m)} c$ and $c^\ast = \exists_{(i \in m)} c$. The constraint $c^\ast$ is the $v$-dedicated part of $c$ since it constrains variables $i$ and $i'$. On the other hand, the constraint $c^\ast$ is the $v'$-independent part of $v$ since it does not constrain $i$ or $i'$. Let us define an operation that removes a variable from a predicate, thus reducing its arity:

$$p(i_1, \ldots, i_n) \circ v = \begin{cases} p(i_1, \ldots, i_n, i_{n+1}, \ldots, i_n) & \text{if } v = i_i \\ p(i_1, \ldots, i_{n-1}) & \text{otherwise.} \end{cases}$$

Let us define $p(a_1, \ldots, a_m) \circ v$ similarly. The transformation

$$Comp'' = \langle p(i) \circ v \mid c^\ast, q(o) \circ v \mid p(i) \circ c, q(o) \in Comp \rangle$$

removes $v$ from a strongly-connected component $Comp$.

Removal of a variable from a strongly-connected component preserves divergent entries but might introduce more divergent entries.

**Proposition 4.7** Let $p_v$ be an entry diverging in $Comp$. Then $p_v$ also diverges in $Comp''$.

**Proof.** Since $p_v$ diverges in $Comp$, there is an infinite resolution

$$p_v(n_0) \rightarrow p_1(n_1) \rightarrow p_2(n_2) \rightarrow \cdots \rightarrow p_k(n_k) \rightarrow \cdots$$

with $p_1(i_1) \rightarrow_c p_2(i_1, a_1) \in Comp$, $p_2(i_1, a_1) \rightarrow_c p_3(i_1, a_1)$ and $\theta$ solution of $c_1[i_1 \mapsto n_1]$. Hence $p_1(i_1) \circ v \rightarrow c^\ast, p_2(i_1, a_1) \circ v \in Comp''$ and $\theta$ is a solution of $c_1^\ast[i_1 \mapsto n_1]$ since $c_1 \models c_1^\ast$. Then $\theta$ is a solution of $c_1^\ast[p_1(i_1) \circ v \rightarrow p_2(n_1) \circ v]$.\hfill $\square$
4.2 Removing unsupported or subsumed clauses

By removing unsupported clauses i.e., clauses that call undefined predicates, we maintain the termination-equivalence of programs, since unsupported clauses cannot be used to build an infinite derivation.

Example 4.3 Let \( P = \{ C_1, C_2, C_3 \} \) with \( C_1 = (\text{entryl}(ix) :- ix = M, q(ax)) \), \( C_2 = (q(ix) :- ix = 1, \text{entryl}(ox)) \) and \( C_3 = (q(ix) :- ix \geq M, r(ax)) \). Predicate \( r \) is not defined in \( P \) and hence clause \( C_3 \) is unsupported. Thus \( P \) is termination-equivalent to \( P' = \{ C_1, C_2 \} \).

Proposition 4.4 (Correctness of unsupported clause removal) Let \( P \) be a program and \( P_1 \) be \( P \) deprived of unsupported clauses. Then \( P \equiv P_1 \).

Proof. Any divergent resolution in \( P_1 \) is also a divergent resolution in \( P \) since \( P_1 \) has less clauses than \( P \). Any divergent resolution in \( P_1 \) is also a divergent resolution in \( P \) since a divergent resolution in \( P_1 \) cannot use any unsupported clause, or otherwise it would be finite.

Another simplification consists in removing subsumed clauses (see also [8]). Let for instance \( C_1 = (p(\xi) :- c_1, q(o)) \) and \( C_2 = (p(\xi) :- c_2, q(o)) \). We say that \( C_2 \) subsumes \( C_1 \) if \( c_1 \models c_2 \) (\( c_1 \) entails \( c_2 \)). Note that \( C_1 \) and \( C_2 \) only differ in the constraint part.

Example 4.5 The program obtained at the end of Subsection 4.2 contains clauses

\[
\begin{align*}
\text{entry3899}(1:\text{L}) & :- \text{L0} \Rightarrow 0, \text{L0} \Rightarrow 1, \text{L0} \Rightarrow 2, \text{entry3899}(\text{L}). \\
\text{entry3899}(2:\text{L}) & :- \text{L0} \Rightarrow 0, \text{L0} \Rightarrow 2, \text{entry3899}(\text{L}).
\end{align*}
\]

The second clause subsumes the first which can hence be removed.

Proposition 4.6 (Correctness of subsumed clause removal) Let \( P \) be a program and \( P_1 \) be \( P \) deprived of subsumed clauses. Then \( P \equiv P_1 \).

The same \(<\text{clr:add}>\) instruction would have been used if two floating point values had been added. The CLR states that the virtual machine has to determine the correct type through some mechanism. This could be accomplished by either a static data flow analysis of the program or by maintaining a type-stack at runtime.

The \(<\text{clr:add}>\) instruction does not check for overflow. If an overflow occurs, only the least significant bytes of that type are considered for the sum of the arguments. If the C# program above had computed the sum using the expression

\[
\text{checked(int)}(a * b);
\]

it would have resulted in the byte code instruction

\[
\text{<clr:add.overflow>} \quad \text{add overflow}
\]

that raises a runtime exception when an overflow occurs. Note that for the \(<\text{clr:add.overflow}>\) instruction the VM also has to determine the correct type of the arguments similar to \(<\text{clr:add}>\).

3 Overview of the JVM and CLR

An exhaustive comparison between the JVM and the CLR is outside of the scope of this paper, as our main interest lies in the exploration of bytecode manipulation with XML technologies. In the following we focus on two key features of the CLR for which there is no corresponding support in the JVM. Later in this paper we will present how these features can be mapped to the JVM. Specifically, we will demonstrate the power and flexibility of using XML, in conjunction with XSL and generics are left for future work. The work involved with this mapping will allow us to perform cross-compilation. The features discussed in the following should allow non-trivial applications to be cross-compiled using a declarative approach. In the following sections we discuss primitive types (Section 3.1) and value types (Section 3.2).

3.1 Numerical Primitive Types

As with any high-level programming language, the languages based on the JVM and the CLR both define a set of primitive types such as bytes, integers, or doubles. Furthermore, both execution platforms define various bytecode instructions such as addition or subtraction that operate on these primitive types. We first discuss the different data models of the JVM and the CLR and then introduce the bytecode instructions.

The JVM supports numerical types of various sizes and precisions. In the following we focus on integer types. Based on the Java programming language, the JVM supports four different integer types, ranging from 8 to 64 bit precision. One interesting fact is that the JVM (and therefore Java) only supports signed integers. I.e., there is no built-in support for unsigned integer types. As with the JVM, the CLR also supports integer types of various precisions. One important difference however is that the CLR supports both signed and unsigned integer types.

Based on the integer types, both virtual machines offer various byte code instructions that operate on those types. In the following we introduce the bytecode instructions for adding two integer values. The bytecode instructions for subtraction, multiplication, and division follow the same pattern. The JVM features the
4 Program Simplifications

4.1 Removing clauses outside loops

In graphs such as that in Figure 2, arrows outside loops cannot be executed during a divergent computation, which stays inside the same strongly-connected component of the entry where it is started (Definition 3.3). Hence it seems reasonable to remove any clause that is not part of a loop i.e., such that its head and tail do not belong to the same strongly-connected component of blocks. For instance, only the second clause in (1) is generated.

The following result formalizes of well-known technique used in many termination analyzers. It allows us to prove termination for the loops of the program. A clause $p(n) \leftarrow c, q(m)$ occurs in a loop if $p$ and $q$ are inside the same strongly-connected component of predicates.

Proposition 4.1 (Correctness of clauses outside loops removal) Let $P$ be a program and $Ps$ the same program deprived of those clauses that do not occur in a loop. Then $P \equiv Ps$.

Proof. We have $Ps \supset P$ since $Ps \subseteq P$. It remains to prove $P \supset Ps$. Programs $P$ and $Ps$ have the same set of entries. Let $q$ be an entry. If $q$ terminates in $P$ then it terminates in $Ps$ since the latter has less clauses than $P$. If $q$ diverges in $P$ then there is an infinite derivation using only predicates inside the strongly-connected component of $q$. Hence only clauses in $Ps$ are used by that derivation, so that $q$ diverges in $Ps$.

If we apply this simplification to the CLP program derived from the Java program in Figure 1, the number of clauses decreases from 297 to 12 and the time needed to prove all the entries terminating is 2.72 seconds.

Because of this simplification, from now on we assume that each predicate is only used in its strongly-connected component. Hence termination according to Definition 3.3 corresponds, from now on, to termination by using all the clauses of the program.

4.2 Removing clauses by unfolding

If a program contains clauses $p(m) \leftarrow c_1, q(m)$ and $q(v) \leftarrow c_2, s(w)$, we can unfold them into the clause $p(m) \leftarrow c_1 \land q \land m = v, s(w)$ (we assume without loss of generality that clauses are renamed so that they do not share variable). If this is done systematically, for all occurrences of $q$ on the right of the clauses of $P$, and the clauses defining $q$ are later removed, we say that we unfold $q$ away from $P$. The result is a program with less predicates but potentially more clauses than $P$. However, subsequent simplifications will usually remove most of them, so that this simplification is useful in practice.

Proposition 4.2 (Correctness of unfolding away of a predicate) Let $P$ be a program and $q$ a non-entry predicate in $P$ with no clause of the form $q(m) \leftarrow c, q(m)$. Let $Ps$ be $P$ where $q$ has been unfolded away. Then $P \equiv Ps$.

The following two different bytecode instructions for adding integers:

- `<jvm:iadd>`: adds two signed 32 bit integers of type int.
- `<jvm:ladd>`: adds two signed 64 bit integers of type long.

Interestingly, there are no special instructions for adding 8 bit and 16 bit integer values. As noted in section 3.11.1 of the JVM specification, “encoding types into opcodes places pressure on the design of [the VM’s] instruction set” [9]. As a consequence, the JVM designers wanted the `<jvm:iadd>` instruction (along with other 32 bit integer-based instructions) to be able to work with both bytes and shorts. This is accomplished by sign-extending values of these types to 32 bit integers when loaded onto the stack.

Despite the fact that an overflow may occur, execution of an `<jvm:iadd>` or `<jvm:ladd>` instruction never throws a runtime exception. The result is the 32 lower-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type int. If an overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

There are only two bytecode instructions for adding integers offered by the JVM, the CLR supports three different op-codes for adding integers:

- `<clr:add>`: adds two signed integers without overflow check.
- `<clr:add_orf>`: adds two signed integer values with overflow check. Throws `OverflowException` if an overflow occurs.
- `<clr:add_orf_un>`: adds two unsigned integer values with overflow check. Throws `OverflowException` if an overflow occurs.

It is interesting to note that the description for the add instruction only specifies the addition of two signed integers. There is no mention of unsigned integers. How can it be possible, then, that the CLR supports the basic addition of unsigned integers? It turns out that the CLR imposes a constraint somewhat similar to that of the JVM for items placed upon its evaluation stack. Only signed 32 bit and 64 bit integers can be loaded onto the stack. Unsigned types are simply sign-extended and treated as signed types when loaded onto the stack. The CLR then takes advantage of the fact that, for some operations, signed and unsigned integers do not need to be treated differently. In those cases where they need to be treated differently, special instructions such as `<clr:add_orf_un>` are used [2].

Based on this examination, we can see that there are several significant differences between the JVM and CLR with respect to their support of integer types. This complicates the efforts involved with cross-compilation. To translate bytecode from the CLR to the JVM, for example, the cross-compilation process must at least provide support for unsigned integers, use the appropriate bytecode instructions for different data types, and check for overflow by throwing a runtime exception when an overflow occurs.
We define now the operational semantics for CLP over linear integer constraints. It expresses the fact that variables stand for the path-length of concrete data structures in the memory of the system and hence can be undefined but not free, in the sense of logic programming.

**Definition 3.2 [Operational Semantics of our CLP Language]** Let \( p, q \) be predicates and \( m, n \in \mathbb{Z}^* \). We say that \( q(n) \) is derived from \( p(m) \) using clause \( C = (p(i) \rightarrow c \cdot q(o)) \), written \( p(m) \vdash^C q(n) \), if there is a solution \( \theta \) of \( C \vdash \theta \) such that \( q(n) = q(o) \theta \). Clause \( C \) in \( p(m) \vdash^C q(n) \) is often omitted unless necessary. A derivation of \( p_0(n_0) \) is \( p_0(n_0) \rightarrow p_1(n_1) \rightarrow \cdots \rightarrow p_s(n_s) \) such that \( p_{i+1}(n_{i+1}) \) is derived from \( p_i(n_i) \) for all \( 0 \leq i < s \). A resolution is a maximal derivation.

The above operational semantics lets us formalise the notion of termination. It uses a partition of the predicates of the program in strongly-connected components. Namely, for every clause \( p(i) \rightarrow c \cdot q(o) \), let \( p \leq q \). Then predicates \( p_0 \) and \( p_1 \) belong to the same strongly-connected component if and only if \( p_0 \leq p_1 \) and \( p_1 \leq p_0 \) where \( \leq ^* \) is the reflexive and transitive closure of \( \leq \). This means that they are part of the same loop. A predicate \( q \) is an entry if it occurs in a clause \( q(n) > c \cdot s(m) \) with \( q \) and \( s \) in the same strongly-connected component (i.e., in a loop) and also in a clause \( t(v) \rightarrow c \cdot q(u) \) with \( q \) and \( t \) in different strongly-connected components. We assume that entries are named \texttt{entry}. From now on, when we say that a predicate is an entry of a CLP program, we mean that its name is \texttt{entry} for some \( x \).

**Definition 3.3 [Termination]** An entry \( p \) terminates in a program \( P \) if, for every \( n \in \mathbb{Z}^* \), all resolutions of \( p(n) \) using the clauses of \( P \), with predicates in the strongly-connected component of \( p \), are finite. Otherwise, \( p \) is said to diverge. Let \( P_1 \) and \( P_2 \) be programs. \( P_1 \) terminates more than \( P_2 \), and we write \( P_1 \triangleright P_2 \), if whenever an entry of \( P_1 \) terminates in \( P_1 \), it also terminates in \( P_2 \). They are termination-equivalent, and we write \( P_1 \equiv P_2 \), if \( P_1 \) terminates more than \( P_2 \) and vice versa.

Note that if \( p \) is not defined in \( P \) then it terminates in \( P \) since its derivations have length 1. The notion of \( P_1 \) terminating more than \( P_2 \) entails that a proof of termination for the predicates of \( P_2 \) is also a proof of termination for the predicates of \( P_1 \).

Definition 3.3 formalizes a loop-local termination. This means that an entry terminates if it terminates by using the predicates of the loop where it occurs. This is important to report a feedback to the user about which loop of which method might introduce the non-termination, without considering entries that diverge just because the computation, after executing the loop where the entry occurs, continues into another loop that diverges. Entries can also be used to improve the precision of the analysis by computing call-patterns from them to the other blocks [17]. We do not discuss this optimization here.

Next section presents a set of program transformations that simplify a CLP program \( P \) into a smaller program \( P_1 \). It will always be the case that \( P \) and \( P_1 \) are termination-equivalent.

### 3.2 Value Types

The CLR introduces the notion of value types. Value types are similar to classes, but their instances are allocated on the stack. Instances of classes (i.e., objects) are usually allocated on the heap and are garbage collected when not used anymore. Garbage collection introduces significant runtime overhead. However, since value types are allocated on the stack, they will be automatically reclaimed when the method where they were defined is exited. For this reason, the underlying behavior of value types in the CLR differs from that of classes. This is demonstrated by the following C# program:

```csharp
using System;
public struct Person {
    public string Name;
    public Person(string name) {
        Name = name;
    }
}

class ValueTypeTest {
    public static void Main() {
        Person p1 = new Person("Bob");
        Person p2 = new Person("Bob");
        assert(p1.Equals(p2));
    }
}
```

In C#, a struct defines a value type (line 4). Although a value type is also instantiated via the \texttt{new} operator (line 14), it will effectivelly be allocated on the stack. As a consequence, variable \texttt{p2} in the example above will copy the value type (line 15). If \texttt{Person} were a proper class, \texttt{p1} and \texttt{p2} would be referencing the same object. But as can be seen by the assertions in lines 17 and 18, \texttt{p1} and \texttt{p2} are separate copies. Value types can be converted to and from proper garbage collected objects. Converting a value type to an object is called boxing, while the reverse mapping is called unboxing. Apart from the boxing and unboxing operation, the CLR introduces no special bytecode instructions for handling value types. The same bytecode instructions are used for objects and value types, but their semantics are different. The JVM offers no support for value types and therefore this requires special handling during cross-compilation.

### 4 CLR to JVM Transformation

The preceding section outlined some differences between the CLR and the JVM. The CLR is a superset of the JVM in many ways, which is underlined by the fact that the CLR supports more primitive types as well as features additional bytecode instructions not found in the JVM. Therefore, it is relatively straightforward to map a JVM program to the CLR. There already exists at least one project that implements this direction of the transformation [4].

The reverse direction of the transformation poses more challenges, and in the following we discuss some techniques to accomplish this with the use of various XML
4.1 Data Flow Analysis

Up to this point we have shown how XML can be used to represent any program that can be executed on the JVM or the CLR. As outlined earlier, the CLR only features un-typed instructions: this is illustrated by our examination of integer addition. As a consequence, it will become necessary to know the type of the operands during the cross-compilation process. Without this knowledge, it would be impossible to map the CLR-add instruction to one of the typed add-instructions supported by the JVM. As a prerequisite, one has to determine on which specific argument types the un-typed instructions operate. This can be accomplished by a data flow analysis. This task is similar to what a bytecode verifier has to do when loading a program into the virtual machine [6].

During a data flow analysis, all the execution paths through a program are traced, but instead of pushing and popping specific arguments onto the stack, only the types of those arguments are stored on the stack. For this reason it is also called a type stack (vs. an argument stack). With this analysis, it is possible to determine the type of the arguments that will be stored on the stack at any point during the execution of the program.

As a first step towards a data flow analysis, we take advantage of XML’s extensibility and introduce new markup of an XMLVM program that can capture the effects an individual bytecode instruction has on the type stack. For each instruction, we need to record the state of the type stack before and after execution of that instruction. Those two states will be marked up with the tags <stack-pre> and <stack-post> respectively and appended as children to the XML tag of the bytecode instruction. The elements contained on a stack are denoted by the <elem>
The catch pseudo-bytecode marks the beginning of a default exception handler which throws back all exceptions to the caller. Bytecodes inside each block are abstracted into a linear constraint $c$ over-approximating the path-length of each local variable and stack element at its beginning and at its end [16]. For instance, for block 6391 we have $c = (100 - q1 = 0, q1 - q4 = 0, q0 - q5 = 0, q1 - q6 = 0, IL0 - qlo = 0, q6 ≥ 0, q62 ≥ 0, IL0 - q63 ≥ 1, IL0 - q62 ≥ 1)$. The variables $q1n$ stand for the path-length of the $n$th stack element at the beginning of the block; $q0n$ for their path-length at the end of the block; $q1n$ and $q0n$ are the same for the $n$th local variable. This constraint is then used to build CLP clauses. In principle, there is a CLP clause for each arrow in the graph of basic blocks. Let block $b$ be a predicate expressing the path-length of the variables in scope at the beginning of block $b$. We build clauses

\[
\text{block6391}(IL0, IL1, IS0) : \neg c, \text{block6392}(IL0, q10, q30, q51, q52, q53, q54).
\]

since two arrows connect block 6391 with blocks 6392 and 6560. Two local variables $L0$ and $L1$ are in scope there ($IL0$ implements this and $IL1$ implements other). At the beginning of block 6391 there is only one stack element $00$, while there are 5 at its end. Those clauses form a CLP program whose termination entails that of the original Java bytecode program [17]. The clauses of that program have exactly one predicate on their right.

Although the program in Figure 1 is relatively small, the number of arrows in its graph of basic blocks is quite large: the resulting CLP program consists of 297 clauses. The aim of the present paper is to introduce simplification techniques for such CLP programs which shorten the termination proofs. Next sections formalize our notion of CLP programs and show how these programs can be simplified.

### 3 CLP over Linear Integer Constraints

We formalize here the CLP programs of the previous section. Namely, they are sets of predicates, each defined by a set of clauses. We require that predicates are named block or entry. Predicates are not distinguished by their arity. That is, two different predicates must be distinct identifiers. For our purposes, clauses arise from arrows in the graph of basic blocks, so we can assume them to have the form $p(x) \cdot c, q(o)$, where $i$ and $o$ are disjoint sequences of distinct variables and $c$ is a linear integer constraint on $i$ and $o$. This is similar to [8] and more general than [3], where binary clauses express size-change graphs, although a more limited form of constraints is used there. Each local variable or stack element $v$ in the bytecode program induces an input variable $i$ and an output variable $o$ in the CLP program. The sequence $v$ consists of only input variables and $o$ of only output variables. For each clause in the program, we refer to three sets of variables $V$, $I$ and $O$: they are the sets of bytecode variables, induced input variable and induced output variables, respectively.

The DFA for this instruction can only be computed once the $<stack-post>$ of the preceding instruction has been determined. This condition is given by the XPath expression of the $\text{match}$ attribute (line 1). The first $<\text{copy-of}>$ copies all attributes to the result tree via the $\&$ expression (line 3). Next, the template defines the type stack before the instruction executes via the $<\text{stack-pre}>$ tag. The state of the type stack at this point is identical with the $<\text{stack-post}>$ of the previous instruction. Therefore, the XPath expression of the second $<\text{copy-of}>$ selects the $<\text{stack-post}>$ of the first preceding sibling of the current instruction (lines 4-6).

The above template finally defines the type stack after executing the $<\text{clr:ldloc}>$ instruction via the $<\text{stack-post}>$ tag (lines 7-15). In case of this instruction, one new element is added to the top of the type stack whose type is determined by the type of the variable whose value is to be pushed onto the stack. The following is an example of the XML template for $<\text{clr:ldloc}>$:

```
<xml template match="clr:ldloc[preceding-sibling::x[1]/dfa:stack-post]">
  <xml:copy-of select="preceding-sibling::x[1]/stack-post/*"/>
  <xml:attribute name="index" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
    name="index"
    select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
  >
    <xml:attribute name="type" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
      name="type"
      select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
    >
      <xml:attribute name="value" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
        name="value"
        select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
      >
        <xml:attribute name="idname" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
          name="idname"
          select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
        >
          <xml:attribute name="type" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
            name="type"
            select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
          >
```

The DFA for this instruction can only be computed once the $<stack-post>$ of the preceding instruction has been determined. This condition is given by the XPath expression of the $\text{match}$ attribute (line 1). The first $<\text{copy-of}>$ copies all attributes to the result tree via the $\&$ expression (line 3). Next, the template defines the type stack before the instruction executes via the $<\text{stack-pre}>$ tag. The state of the type stack at this point is identical with the $<\text{stack-post}>$ of the previous instruction. Therefore, the XPath expression of the second $<\text{copy-of}>$ selects the $<\text{stack-post}>$ of the first preceding sibling of the current instruction (lines 4-6).

The above template finally defines the type stack after executing the $<\text{clr:ldloc}>$ instruction via the $<\text{stack-post}>$ tag (lines 7-15). In case of this instruction, one new element is added to the top of the type stack whose type is determined by the type of the variable whose value is to be pushed onto the stack. The following is an example of the XML template for $<clr:ldloc>$:

```
<xml template match="clr:ldloc[preceding-sibling::x[1]/dfa:stack-post]">  
  <xml:copy-of select="preceding-sibling::x[1]/stack-post/*"/>
  <xml:attribute name="index" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
    name="index"
    select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
  >
    <xml:attribute name="type" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
      name="type"
      select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
    >
      <xml:attribute name="value" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
        name="value"
        select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
      >
        <xml:attribute name="idname" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
          name="idname"
          select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
        >
          <xml:attribute name="type" select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
            name="type"
            select="preceding-sibling::*[1]/dfa:stack-post/*[preceding-sibling::*[1]]/stack-pre/*[preceding-sibling::*[1]]/clr:ldloc*"
          >
```
The next step of the XMLVM toolchain consists in translating the CLR instructions to JVM bytecode. In some cases this mapping is trivial, as there is a one-to-one correspondence between CLR and JVM instructions. One example is the <clr:add> instruction, which pushes a null reference onto the execution stack. This can be directly mapped to the <jvm:aconst_null> instruction. In other cases, the mapping has to rely on the data flow analysis as introduced in the previous section. In the following sections we demonstrate how to map CLR bytecode instructions for numerical operations and value types to semantically equivalent JVM bytecode instructions.

### 4.2.1 Mapping Numerical Operations

As explained earlier, the CLR only features un-typed instructions, whereas the JVM only has typed instructions. The <clr:add> instruction of the XMLVM.CLIR.DFA excerpt above it can be seen that the data flow analysis added tags for each CLR instruction that reflects the content of the type stack before and after the execution of that instruction. As shown in lines 17 to 25, the CLR instruction <clr:add> pops off two integers and pushes an integer back onto the type stack. It should be noted that the data flow analysis done here is much less complex than what a bytecode verifier typically is required to check. In particular, since we are only interested in primitive types, it is not necessary to compute the LUB (least upper bound) of object types of different execution paths [8].

### 4.2 Generating JVM Bytecode

In the example Java program.

```
public class List<X> {
    private X head; private List<X> tail;
    public List(X[] values) { this(values,0); }
    public ...
    ...
}
```

In the XMLVM.CLIR.DFA excerpt above it can be seen that the data flow analysis added tags for each CLR instruction that reflects the content of the type stack before and after the execution of that instruction. As shown in lines 17 to 25, the CLR instruction <clr:add> pops off two integers and pushes an integer back onto the type stack. It should be noted that the data flow analysis done here is much less complex than what a bytecode verifier typically is required to check. In particular, since we are only interested in primitive types, it is not necessary to compute the LUB (least upper bound) of object types of different execution paths [8].

```
<p>public class List<X> {</p>
<p>private X head; private List<X> tail;
public List(X[] values) { this(values,0); }
public ...
public ...
public ...
}<p>
</p>
```

new (Subsection 4.4). Moreover, we present all such simplifications together and prove them correct in a uniform setting, which was not the case before. Furthermore, we experiment with their effects on the termination analysis of real, large software, which was never the case before; in particular, those simplifications have never been applied to the termination analysis of Java bytecode.

### 2 Our Running Example

Consider the Java program in Figure 1. It implements a generic list of elements of type X. Two constructors are available. The first builds a list from head and tail, and the second builds recursively a list from an array. The method append concatenates two lists this and other. The method afterInteger writes null after all elements of the lists of type Integer. Method toString() yields a String representing the list elements as asterisks, but does not represent the elements that follow an object of type Integer. All these methods are recursive. Method main builds some lists and calls the previous methods.

We compile this program into Java bytecode and analyse the bytecode as in [17]. Our system tells us that the program terminates. We refer to [17] for the detailed description of how our system works. Here, we briefly give an intuition. First, the Java bytecode is transformed into a graph of basic blocks [1], as done in Figure 2 for method append. Recursion is made explicit by linking each method call to the beginning of the called method(s), as we do for block 6560 in Figure 2. The makescope pseudo-bytecode creates the activation stack for a method with arguments of type
to software downloaded from insecure networks into computers or cellular phones: the user wants a proof that that software will actually terminate and yield a result or otherwise he will not use it and pay for it.

Termination analyses have been developed for logic \cite{8,10,7}, functional programs \cite{14} and term rewrite systems \cite{11}, whose semantics is relatively simple and well understood. More recently, termination analysis has been applied to imperative programs, dealing with primitive values only \cite{9,15}, lists \cite{13,6,5,4}, or any dynamic data-structure \cite{17}. In all cases, termination is typically proved by showing that some well-founded measure decreases along loops and recursion, so that divergence cannot occur. This measure can be the value of a variable of primitive type, the length of a list, the maximal path of pointers reachable from a given variable \cite{16} or a mix of such values. When generic data structures are considered, the shape of the computer memory must be somehow approximated, since destructive updates mute dynamic data through shared pointers. Possibly cyclical data structures must be detected, since iterations over them might diverge.

In \cite{17}, a termination analysis is defined working for any sequential Java bytecode program \cite{12}, dealing with any dynamic data structure, possibly cyclical and shared. Since Java is compiled into Java bytecode, that technique can also be used for termination analysis of Java. It works by translating the Java bytecode program into a constraint logic program (CLP) expressing size relationships between program variables at different program points. It has been proved in \cite{17} that if the CLP program terminates then the original Java bytecode program terminates. Hence all techniques for termination analysis of CLP can be used to prove termination of Java and Java bytecode. In \cite{17}, the BinTerm termination prover is used to that purpose. Experiments scale to programs of up to 1000 methods. Although this is already an impressive result, it must be acknowledged that the analysis is expensive in terms of the time needed to build the proof of termination.

In this paper we contribute to the termination analysis of Java and Java bytecode programs. Namely,

- we present a set of simplifications of the CLP programs generated by the termination analysis in \cite{17}. They transform the program by removing clauses or variables, yet preserving its behaviour w.r.t. termination;
- we prove those transformations correct;
- we experiment with those transformations and show them effective: they reduce by orders of magnitude the cost of finding a termination proof for the CLP programs.

These techniques are now embedded in the termination prover for Java bytecode available at the address \url{http://julia.science.univr.it/termination}.

Although some of our simplifications are, often implicitly, used in the termination analysis of programs, this is not the case for others. Namely, the restriction to only those clauses that form a loop in the code (Subsection \ref{subsec:loop}) cannot be applied to other frameworks, such as the termination analysis of logic programs, since one needs the removed clauses there, in order to take care of instantiation patterns due to the presence of logical variables (which do not exist in our setting). Also the simplifications based on removing variables which are irrelevant for termination are

```
public class MathLib {
    static public int add_ovf(int x, int y) {
        int z = x + y;
        if (z == ((long) x + (long) y))
            return z;
        else
            throw new OverflowException();
    }
}
```

This above XSL template demonstrates that bytecode instructions can be very easily inlined. The JVM tag `<jvm:invokestatic>` serves the same purpose as the CLR tag `<clr:call has-this="false"/>` for static methods. Whenever the `<clr:add_ovf>` instruction is encountered in a CLR program, an invocation to a compatibility library is made that mimics the semantics of that instruction. The following Java class MathLib shows one possible implementation of the `<clr:add_ovf>` bytecode instruction for integers for which no direct correspondence exists in the JVM. Note that the signature of this method is consistent with the `<clr:add_ovf>` instruction:

```
public class MathLib {
    public static int add_ovf(int x, int y) {
        int z = x + y;
        if (z == ((long) x + (long) y))
            return z;
        else
            throw new OverflowException();
    }
}
```
Using CLP Simplifications to Improve Java Bytecode Termination Analysis

Fausto Spoto
Dipartimento di Informatica
Università di Verona
Italy

Lunjin Lu
Oakland University
USA

Fred Mesnard
LIM IERMA
Université de la Réunion
France

Abstract

In an earlier work, a termination analyzer for Java bytecode was developed that translates a Java bytecode program into a constraint logic program and then proves the termination of the latter. An efficiently executable terminating logic program can then be obtained in the desired computer language. However, the termination proof for the constraint logic program depends on many details of the Java bytecode and any simplifications of the latter are introduced to ease the termination analysis. In this paper, a set of program simplifications are presented that reduce the size of the constraint logic program without changing its termination behavior. These simplifications remove program clauses and/or predicate arguments that do not affect the termination behavior of the constraint logic program. Their effect is to reduce significantly the time needed to build the termination proof for the constraint logic program, as our experiments show.

Keywords: Java, Java bytecode, static analysis, termination

1 Introduction

Termination analysis attempts to prove that programs terminate. Since termination of Turing-equivalent programming languages is undecidable [18], termination analysis only succeeds for a (hopefully large) class of programs, although many terminating programs are not proved to terminate. Despite this limitation, it is increasingly important in software technology, since proofs of termination add value

4.2.2 Mapping Value Types

Value types have been introduced in the CLR as lightweight objects for which the JVM offers no equivalent bytecode operations. The following XMLVM shows the CLR bytecode instructions for lines 14-15 of the ValueTypeTest.Main() method introduced in Section 3.2:

```xml
<vm:signature>
  <vm:parameter type="String"/>
  <vm:return type="void"/>
</vm:signature>
```

The <clr:var> declarations of type Person will allocate sufficient memory on the stack to hold instances of that value type. Since value types are allocated on the stack, they are not created via <clr:newobj>. The <clr:ldloc> (load location address) pushes the address where the value type is allocated onto the stack. The following <clr:call> instruction then calls the constructor (.ctor) of the value type. The combination of <clr:ldloc> and <clr:stloc> copies a value type. Note that those two bytecode instructions also work with regular objects, but since they are applied to value types in this example, a deep copy is performed. Analyzing the bytecode instructions created by a C# compiler, several observations can be made:

- Value types are not allocated via the instruction <clr:newobj> that is typically used to instantiate a new object on the heap.
- Value types are allocated on the stack. The <clr:var> variable declaration implicitly allocates memory for the new value type on the stack.
- Value types are manipulated via the same bytecode instructions as regular objects. The <clr:stloc> instruction in this case does a deep copy of a value type.

Since the JVM has no support for value types, they have to be simulated while retaining the semantics as defined by the CLR. The easiest approach is to convert value types to heap-allocated objects. If value types are mapped to objects, they need to be allocated via <clr:newobj>. Special care has to be taken when copying value types. The <clr:ldloc> and <clr:stloc> copy references for objects, while performing a deep copy for value types. In this case, the data flow analysis again helps to map these bytecode instructions to proper JVM instructions. Since a deep copy cannot be performed with a single bytecode instruction, we generate a call to a compatibility library that implements this behavior. Each value type inherits directly or indirectly from class System.ValueType. We add a static method _CpV to this class that performs the deep copy using the Java reflection API. Here is the declaration of this method:

```java
public class ValueType extends System.Object {
```
With the assistance of this helper method, the bytecode instruction `<clr:stloc>` will be mapped to a static invocation of this method whenever the instruction is handling a value type:

```csharp
static public void __COPY(ValueType from, ValueType to)
{
    ...
}
```

The `<clr:stloc>` instruction expects the value type to be stored on the stack. Since we treat all value types as objects, this value type will be represented as a reference. The destination (also represented as a reference) is pushed onto the stack via `<jvm:aload>` before making a call into the compatibility library. The signature of method `__COPY` is chosen such that it matches the stack layout at this point in time.

### 4.3 API Transformation

Cross-compiling CLR bytecode instructions to JVM bytecode instructions does not solve the problem of external libraries referenced by the application. E.g., if a C# application uses Windows Forms to create a button on a user interface, it will reference class `System.Windows.Forms.Button`. Cross-compiling bytecode instructions would result in a Java class file having an external reference to this class that does not exist in the Java runtime library. One solution is to create a set of compatibility classes in Java with the exact API as their CLR counterparts.

In some cases it is possible to map the API without the need of compatibility classes as can be seen with the .NET class `System.String`. Its behavior is almost identical to that of the Java class `java.lang.String`. In principle, every reference of `System.String` can be replaced with `java.lang.String` in the cross-compiled program. There are some important differences, however, in their respective implementations. One such difference is the way the length of a string is determined.

```csharp
namespace System {
    class String {
        public int Length { get; }
        // ...
    }
}
```

When used in a C# program, the resulting bytecode calls an instance method of name `get_length()`. This method has the same behavior as `java.lang.String.length()` and it can therefore be replaced. This is accomplished by the following XSL template:
Changing class names and method names is the essence of API mapping. When used together with API wrapping, the cross-compiled Java application behaves identically to its original CLR version. The above XSL template demonstrates the full potential of declarative XPath expression to filter out the desired API.

5 Conclusion and Outlook

In this paper we have demonstrated the feasibility of using XML technologies to perform bytecode manipulations—such as with the cross-compilation of CLR bytecode instructions to the JVM. This allows .NET developers to deploy their applications on a standard JVM. The CLR offers a wider range of features than the JVM, thereby making the cross-compilation non-trivial. We see our work as a showcase for the power of XML technologies. In particular, XSL stylesheets have proven to be a powerful abstraction for bytecode manipulations. XSL is a declarative, Turing complete language that allows us to focus on performing bytecode transformations without having to deal with byte code manipulation libraries such as BCEL or the .NET reflection API.

In this paper we have shown how to map integer operations and value types to the JVM. Other features of the CLR remain for future work. In particular the support for generics will require significant work. Other future research will investigate the Java bytecode optimization as an abstraction for this heterogeneous mix of bytecode instructions.

References

implementation, as well as approximation by collapsing recursions, make the analysis scale. As shown in [5], there are usually rich and large loops within the call graph, and thus much precision is lost by collapsing loops.

SPLAT[23] is a widely-used testbed for experimenting with points-to analysis for Java. It supports both equality and subset-based analysis, provides various algorithms for call graph construction (such as CHA, RTA, and an on-the-fly algorithm), and enables variations on field-sensitivity. The BDD-based implementation of the subset-based algorithms further improves the efficiency of operations on points-to sets [24]. Our analysis also borrows its CHA for a preliminary call graph. A recent empirical study compares precision of subset-based points-to analyses with various abstractions on context-sensitivity [5].

One stream of research examines calling contexts in terms of sequences of objects on which methods are invoked, called object-sensitivity [20]. Similar to call-site strings based approach, the sequence of receiver objects can be unbounded and demands proper approximations, like k-CFA [6]. [5] also concludes that a context-sensitive points-to analysis in terms of object-sensitivity excels at precision and is even more likely to scale by experimental studies. Concerning scalability for context-sensitive points-to analysis, some analysis utilizes BDD as the underlying data structure [23,19], others only compute results that sufficiently meet the client’s needs, so-called client-driven and demand-driven manner [4,18]. These strategies are also applicable to our analysis in this paper.

6 Conclusions

This paper presents context-sensitive points-to analysis for Java as all-in-one weighted pushdown model checking. The notion of valid paths are enriched such that dataflow along each valid path need further satisfy type requirements for dynamic dispatch. The ahead-of-time analysis is formalized as one run of weighted pushdown model checking, which enjoys context-sensitivities regarding both call graph and valid paths. The proposed points-to analysis is implemented as a prototype, with Soot as the preprocessor from Java to Jimple and Weighted PDS library as the model checking engine.

The time complexity in general case specific to our analysis is $O(|\Delta| \cdot |B| \cdot |T_0| \cdot |T_1|)$. $|\Delta|$ is the cardinality of the weight space. $|\Delta|$ is up to the program size by encoding. $|T_0|$ and $|T_1|$ are the prices for each weight operation. At present, the tentative experiments are restricted to small examples, due to the weight package is implemented based on linked list for a fast prototyping. Our next step is to prepare a weight package based on CrocoPat [26], a high level BDD package.

References


5 Related Work

Points-to analysis for Java has been an active field over the past decade. We limit our discussion primarily to recent advances especially related to context-sensitive points-to analysis.

One of the pioneer work in this field is Andersen’s points-to analysis for C [13]. It is a subset-based, flow-insensitive analysis implemented via constraint solving, such that object allocations and pointer assignments are described by subset constraints, e.g. $x = y$ induces $\text{pta}(y) \subseteq \text{pta}(x)$. The scalability of Andersen’s analysis has been greatly improved by more efficient constraint solvers [14,15]. Andersen’s analysis was introduced to Java by using annotated constraints [16].

Reps, et al. present a general framework for program analysis based on CFL-reachability [11]. A points-to analysis for C is shown by formulating pointer assignments as productions of context-free grammars. Borrowing this view, Sridharan, et al. formulated Andersen’s analysis for Java in a demand-driven manner [17]. The analysis targets on applications with small time and memory budgets. A key insight of their algorithm is that a field read action is supposed to be preceded by a field write action, so-called balanced-parentheses problem. An improved context-sensitive analysis is later proposed by refining call paths as a balanced-parentheses problem as well [18]. The lost precision is retained by further refinement procedures. The demand-driven strategy, as well as the refinement-based algorithm makes this analysis scale.

A scalable context-sensitive points-to analysis for Java is presented in [19]. Programs and analyses are encoded as the set of rules in logic programs Datalog. The context-sensitivity is obtained by cloning a method for each calling context, and by regarding loops as equivalent classes. The BDD (Binary Decision Diagram) based
Lemma 4.17 For $w_1, w_2 \in \mathbb{D}$, and $(\tau, c) \in w_1, (\tau', c') \in w_2$, $henv \in henv_{abs}$, and $(s, t \uparrow m) \in \mathcal{E}$:

$$
\bigcup_{s' \in \text{trace}(\tau, s)} \text{eval}^{\text{abs}}(henv, s') \supseteq \text{eval}^{\text{abs}}(\tau(henv), s)
$$

Lemma 4.17 says that, the result of back-tracing by trace soundly comprise all the contributed path constraints. As illustrated in Figure 2, the analysis is performed on an operational transition sequence, where $\bullet \rightarrow \bullet$ represents an operational transition in one step, and $w_1$ and $w_2$, respectively, denote the resulting weight by composing transitions marked with the dotted line. By Lemma 4.17, to check the points-to targets of $s$ on $\tau(henv)$ amounts to check that of all $s' \in \text{trace}(\tau, s)$ on $henv$.

Theorem 4.18 (Soundness) For $r \in V_{\text{set}}$ at $l \in \text{Loc} \cup \text{Ret}$ and $C = \langle l, (\text{Ret})' \rangle$, $\text{pts}(r, C)$ is sound.

4.3 Prototype Implementation

We implemented the analysis algorithm as a prototype in the Soot framework. As shown in Figure 3, it starts off preprocessing from Java sources (or bytecode) to Jimple codes by Soot. Soot provides facilities of call graph construction and points-to analysis at various levels of precision. We borrow the most imprecise analysis CHA (Class Hierarchy Analysis) [21] to produce a preliminary call graph for the ahead-of-time analysis. A weighted pushdown system designed for the ahead-of-time analysis is then constructed from the Jimple code. The analysis is finally performed by calling the weighted PDS Library on the model, during which the invalid control flows are removed from the analysis results on-demand. Note that, during the encoding of programs as a weighted pushdown system, extra variables will be introduced in RefVar to denote formal parameters and return values of reference types. For program statements whose execution does not change heap states, their corresponding heap environment transformers are thus identity function $henv \cdot henv$, such as the conditional branching statement.

Definition 4.15 defines rules for judging whether a type $t$ conflicts or compatible with a type $r(t)$. For simplicity at the first stage, the case (ii) is not provided in the prototype implementation and will be included in the later version. The right-hand-side of Figure 3 shows the points-to result of analyzing Example 4.1. The analysis returns two abstract heap environment transformers. $f_1$ gives the precise dataflow summary of this program following the control flow $\langle 4 \rightarrow 9 \rightarrow 15 \rightarrow 6 \rightarrow 9 \rightarrow 20 \rangle$, which precisely infers that $\langle a^1 \rightarrow a^3, d^3 \rightarrow a^2 \rangle$. $f_0$ is a dataflow summary of the invalid call path $\langle 6 \rightarrow 9 \rightarrow 15 \rangle$ due to excluding the case (ii) above.

The S3MS.NET Run Time Monitor

Lieven Desmet1 Wouter Joosen1 Fabio Massacci2
Katsiaryna Naliuka2 Pieter Philippaerts1,3 Frank Piessens1
Dries Vanoverbergh1

Abstract

This paper describes the S3MS.NET run time monitor, a tool that can enforce security policies expressed in a variety of policy languages for .NET desktop or mobile applications. The tool consists of two major parts: a bytecode inliner that rewrites .NET assemblies to insert calls to a policy decision point, and a policy compiler that compiles source policies to executable policy decision points. The tool supports both singlethreaded and multithreaded applications, and is sufficiently mature to be used on real-world applications.

Keywords: security, bytecode rewriting, .NET, MSIL

1 Introduction

In today's networked world, code mobility is ubiquitous. Even mobile phones and Personal Digital Assistants increasingly support the installation of third-party applications from a variety of sources. This support for applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications can lead to denial of service, financial damage, leaking of confidential information and so forth. The research community has developed a variety of countermeasures for addressing the threat of untrusted mobile code. One important class of countermeasures addresses this risk by monitoring the application at run time, and aborting it if it violates a predefined security policy.

This paper reports on a tool developed within the Security of Software and Services for Mobile Systems (S3MS) project. It implements a monitor for the .NET platform through bytecode inlining. Several of the key algorithms implemented in the tool have been proven formally correct, and the implementation is sufficiently

---

1 DroxNet Research Group, Department of Computer Science Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium
2 Department of Information and Communication Technology, Università di Trento, Via Sommarive 14, 38123 Trento (Trento), Italy
3 Email: Pieter.Philippaerts@kuleuven.be

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

---
mature to handle real-world applications for both the .NET Compact Framework (for mobile devices) and the full .NET Framework (for desktops and servers).

2 Tool Architecture

2.1 Overview

As case studies [10] show, the major security concerns of users about third-party applications are invocations of functionality that incurs a monetary cost, and treatment of sensitive data. Access to this functionality as well as to the sensitive information is provided by calling system API methods. A simple way to prevent the application from causing harm is to suppress the calls to the potentially dangerous methods, effectively sandboxing the application. However, in this way the useful functionality that the application can provide is also hampered. To allow this functionality to the application, without compromising security, the access to the sensitive system calls (later called security-relevant methods) should be regulated by the policy, which grants access to security-relevant methods according to specified rules. These rules can include conditions on the environment (e.g. time) or on the previous access requests of the application (as in history-based access control [4]).

To define what functionality is considered security-relevant, we rely on the policy.

For example, if the policy prohibits network accesses after sensitive information was accessed by the application, then the security-relevant API calls are “starting a connection” and “accessing sensitive information”. All other operations, such as creating files, are irrelevant to this policy and need not to be monitored. Note, that some operations are more likely to be listed as security-relevant than others. For instance, GUI operations are unlikely to be listed as security-relevant by any realistic policy, and therefore will be executed without any monitoring overhead.

The SMS.NET run time monitor consists of two key components (Figure 1). The unliner rewrites potentially dangerous applications. It scans the bytecode to find security-relevant API calls, and wraps additional code around such calls. This additional code checks whether the application is allowed to perform this call. If so, the wrapper code will silently allow the application to continue. If not, the application will be interrupted.

The second component, called the policy compiler, generates an executable version of the policy that the user has created. The dotted line in Figure 1 signifies that the wrapper code inserted by the inliner will call functions in the executable policy, generated by the policy compiler.

![Figure 1. The architecture of the tool](image)

union extension on maps for all references; and (ii) whether the ordering of the composition of heap environment transformers is kept on a sequence of program codes. Apart from (i), the ⊕ operation on weights \(w_1, w_2\) is extended as \(w_1 \oplus w_2 = \{\text{abs} \circ \text{henv} \circ p_0 \circ \text{henv} \circ p_1 \mid p_0, p_1 \in \text{w_1, w_2}\}\).

Definition 4.12 For a program starting with the entry point \(l_0 \in \text{Loc}\), let \(W = (P, \Sigma, f)\) be the weighted pushdown system encoded from it by Definition 2.6, and let \(\text{Ret}\) be the set of return points introduced for method invocations. The points-to analysis on the reference \(r \in V_{ref}\) at the program point \(l \in \text{Loc} \cup \text{Ret}\) is defined as

\[
\text{pta}(r, C) = \text{eval}_{abs}(\delta(r, C)(\text{henv}), r)
\]

where \(\delta(r, C)\) is from Definition 2.5 with \(r = (\cdot, l_0)\) and \(C = (\cdot, l.(\text{Ret})^+)^t\).

We take \(C = (\cdot, l.(\text{Ret})^+)^t\) to represent all possible pushdown configurations as an approximation, when \(l\) is the top-most stack symbol. Therefore, pta computes points-to-information along all paths leading from the program’s entry point to the program point \(l\) of concern.

4.2 Soundness

Since \(\oplus\) operation conservatively combines all possible dataflow in the analysis, we turn to the following two steps to show that our analysis is so und (Theorem 4.18), (i) the analysis on any sequential execution path infers sound points-to results based on abstract interpretation (Theorem 4.16), and (ii) if some control flow is removed during the analysis, it is invalid indeed in the concrete execution, which is witnessed by Lemma 4.17.

Definition 4.13 Let type : \(\mathcal{O} \rightarrow \mathcal{T}\) and loc : \(\mathcal{O} \rightarrow \mathcal{Loc}\) be functions that return the type and the allocation site of a heap object, respectively. The abstraction on heap objects \(\alpha : \mathcal{O} \rightarrow \mathcal{O}_{\beta}\) is defined as follows,

\[
\begin{align*}
\alpha(\alpha) & = (l, I) \in \mathcal{O} \setminus \{T_0, \perp\}, \text{type}(\alpha) \in \mathcal{T}, l = \text{loc}(\alpha) \in \mathcal{Loc}; \\
\alpha(T_0) & = T_{\alpha}, \text{type}(\alpha) = \perp_{\alpha}, \text{loc}(\alpha) = \perp_{\alpha}
\end{align*}
\]

The concretization is denoted by \(\gamma = \alpha^{-1} : \mathcal{O}_{\beta} \rightarrow \mathcal{O}\). The powerset extensions of \(\alpha\) and \(\gamma\) are denoted by \(\alpha_{\text{abs}}, \mathcal{P}(\mathcal{O}) \rightarrow \mathcal{P}(\mathcal{O})\) and \(\gamma_{\text{abs}}, \mathcal{P}(\mathcal{O}) \rightarrow \mathcal{P}(\mathcal{O})\).

Definition 4.14 For the program entry \(l_0\) and the program point \(l \in \text{Loc} \cup \text{Ret}\), let \((\cdot, l_0, \text{henv})^\leftarrow (\cdot, l, \text{henv})\). For \(r \in V_{ref}\) at \(l\) and \(C = (\cdot, l.(\text{Ret})^+)^t\), \(\text{pta}(r, C)\) is sound if \(\alpha_{\text{abs}}(\text{eval}_{abs}(\text{henv}, r)) \subseteq \text{pta}(r, C)\).

Definition 4.15 For abstract environment transformers \(f_1, f_2 \in \text{ExpFun}, x \in V_{ref}\) and \(\text{henv} \in \text{HenvAbs}, f_1 \triangleright f_2 \text{ if } \text{eval}_{\text{abs}}(f_1(\text{henv}), x) \supseteq \text{eval}_{\text{abs}}(f_2(\text{henv}), x)\).

Theorem 4.16 For a Jimple statement \(s \in \text{Stmt}\), let \(f\) be the heap environment transformer of \(s\), and \(f_{\text{abs}}\) be the abstract heap environment transformer of \(s\). Then, \(f_{\text{abs}} \triangleright f \circ f_{\text{abs}}\).

Theorem 4.16 is proved by a case analysis on the Jimple statement \(s\).
Definition 4.7 defines judgements on path constraints when composing the extended weights.

- The first case says, error returns if the current abstract heap environment is not set to the path constraints on s. This case results in the weight $0$ and the related control flow is thus excluded from the analysis result.
- The second case says, a known satisfied constraint will not be included into the newly generated path constraints for efficiency.
- The last case says, new path constraints are generated when the judgement on path constraints is pending at the moment.

Definition 4.8 Define a semiring $S = \{D_0, \oplus, \otimes, 0, 1\}$, such that
- $D_0 = \mathcal{P}(\mathcal{D})$, where $\mathcal{D} = \{(f, c) \mid f \in \text{ExpFun}, c \subseteq \text{PathCons}\}$
- $0 = \emptyset$ and $1 = \{(henv, henv, \emptyset)\}$
- $w_1 \oplus w_2 = \{p_1 \oplus p_2 \mid p_1 = (\text{func}_1, c_1) \in w_1, p_2 = (\text{func}_2, c_2) \in w_2\}$
- $\{\text{func}_1, c_1\} \otimes \{\text{func}_2, c_2\} = \emptyset$ if $\text{func} = \text{error}$
- $\{\text{func}_1 \circ \text{func}_2, c_1 \cup \text{func}_1\}$ otherwise

where $\text{func} = \text{judge}(c_2, \text{func}_2)$, $w_1, w_2 \subseteq D_0$.

- $w_3 \oplus w_2 = w_1 \cup w_2$ for $w_1, w_2 \subseteq D_0$.

Remark 4.9 Both the associativity of $\oplus$ and the distributivity of $\oplus$ over $\otimes$ hold. Since the nesting of field references and array structures is finite yet unbounded, a bound can be given on their nested depth for efficiency. That is, a field or array reference nested deeper than the given bound will be regarded as pointing to anywhere (i.e., $\top$), as illustrated in Example 4.10. As a result, $(w_0 \oplus w_1) \oplus w_2 \subseteq w_0 \oplus (w_0 \oplus w_2)$ for $w_0, w_1, w_2 \subseteq \mathcal{D}$.

Example 4.10 If we limit the nesting of field references to the depth 1, the analysis of the Java code fragment “x = f = w; y = x; f; z = y; g;” returns the weight $\lambda \text{henv. henv} \cdot \text{henv}(z) . f \mapsto \text{henv}(w) \ast \{z \mapsto \text{henv(henv(w))}\}$ by $(w_0 \oplus w_1) \otimes w_2$, and $\lambda \text{henv. henv} \cdot \text{henv}(z) . f \mapsto \text{henv(henv(w))}\ast \{z \mapsto \text{henv}_{\top}(\top)\}$ by $w_0 \oplus (w_0 \otimes w_2)$.

Remark 4.11 The points-to analysis presented above is flow-sensitive. It is easy to obtain parameterized flow-sensitivity as a hierarchy by loosen the following dimensions in the weight space design, (i) whether the points-to target of a reference is changed by a new assignment on it. For this purpose, is is reinterpreted as the

The tool supports multiple platforms – most notably the .NET Compact Framework and the .NET Full Framework – and hence both tool components additionally take a platform description as input.

We discuss each of the two components in some more detail.

2.2 The Inliner

The inliner loops over the bytecode of an untrusted application looking for calls to security-relevant methods (SRM). Identifying such calls statically in the presence of dynamic binding and delegates (a form of type safe function pointers supported by the .NET virtual machine) is non-trivial. The tool implements the algorithm by Vanoverbergh and Pessemis [8].

Before and after each call to a SRM, a call to the executable version of the policy, called the policy decision point (PDP), is injected. A PDP is a Dynamic Link Library that manages the security state associated with the application. It can be thought of as an implementation of a security automaton [6] that reacts to the start and return (both normal and exceptional) of SRMs.

Listings 1 and 2 show the effect of inlining on a simple program that sends an SMS.

Before each SRM call, a ‘before handler’ is added, which checks whether the application is allowed to call that method. If not, an exception is thrown. This exception will prevent the application from calling the method, since the program will jump over the SRM to the first suitable exception handler it finds.

Likewise, after SRM call, an ‘after handler’ is added. This handler typically only updates the internal state of the PDP. If an exception occurs during the execution of the SRM, the ‘exceptional handler’ will be called instead of the ‘after handler’. In summary, the different handler methods implement the reaction of the security automaton to the three types of events: calls, normal returns and exceptional returns of security-relevant methods.
2.2.1 Inheritance and Polymorphism

The simplified code shown above does not deal with inheritance and dynamic binding. Support for this was implemented by extending the logic in the PDL to consider the type of the object at runtime, instead of only looking at the static type that is available during the inlining process. When a security-relevant virtual method is called, calls are inlined to so-called dynamic dispatcher methods that inspect the runtime type of the object and forward to the correct handler. The details, and a formal proof of correctness of this inlining algorithm is presented in [8].

2.2.2 Multithreading and Synchronization

Inlining in a multithreaded program requires synchronization. Two synchronization strategies are possible: strong synchronization, where the security state is locked for the entire duration of a SRM call, or weak synchronization where the security state is locked only during execution of the handler methods.

Our tool implements strong synchronization, which might be problematic when SRMs take a long time to execute, or are blocking (e.g. a method that waits for an incoming network connection). To alleviate this problem, the tool partitions the handler methods according to which security state variables they access. Two partitions that access a distinct set of state variables can be locked independently from each other.

2.3 The Policy Compiler

The policy compiler is the component that translates source policies, written by the user or the system administrator, into an executable policy decision point.

The tool supports two different policy languages, one that represents security automata by means of an explicit declaration of the state space, and guarded commands that operate on this state, and another one that is a variant of a temporal logic. Both languages extend history-based access control by introducing the notion of scopes. A scope specifies whether the policy applies to (1) a single run of each application, (2) saves information between multiple runs of the same application or (3) gathers events from the entire system.

2.3.1 ConSpec

ConSpec ([]) is directly based on the notion of security automata, and is similar to Erlingsson’s PSLang [7] policy language. Like PSLang, a ConSpec specification includes the definition of state variables and the definition of what state transitions are caused by each of the security relevant events. An SRM is executed if the state allows it, and the state is updated accordingly before or after the execution of the SRM. ConSpec extends PSLang with support for multiple scopes.

2.3.2 2D-LTL

An alternative to ConSpec is the 2D-LTL policy language [5], a temporal logic language based upon a bi-dimensional model of execution. One dimension is a sequence of states of execution inside each run (session) of the application, and another one is formed by the global sequence of sessions themselves ordered by

---

Table 4

<table>
<thead>
<tr>
<th>Abstract Heap Environment Transformer with Path Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.m(r0, ..., rl) (τ0 ◦ τ1 ◦ ... ◦ τk, {(x, t ↑ ... t ↑ m)}, τ) = error∪</td>
</tr>
<tr>
<td>(s,t↑m)∈c judge({(s, t ↑ m)}, τ) otherwise</td>
</tr>
</tbody>
</table>

The evaluation function eval\text{abs} is extended as eval\text{abs}: Hen\text{abs} → V ∪ Obj → P(Obj), such that for henv ∈ Hen\text{abs}, o ∈ Obj

$$\text{eval}^{\text{abs}}(\text{henv}, o) = \{o\}$$

$$\text{eval}^{\text{abs}}(\text{henv}, v) = \text{henv}(v)$$

$$\text{eval}^{\text{abs}}(\text{henv}, V, f) = \{\text{henv}(o, f) | o ∈ \text{eval}^{\text{abs}}(\text{henv}, V)\}$$

$$\text{eval}^{\text{abs}}(\text{henv}, \{V\}) = \{\text{henv}(o) | o ∈ \text{eval}^{\text{abs}}(\text{henv}, \{V\})\}$$


$$\text{Ref}(\tau, v) = \begin{cases} \{vt\} & \text{if } vt = v \notin \text{Arf} \\ \text{Ref}(\tau’, v) ∪ \{vt\} & \text{if } vt = v ∈ \text{Arf} \\ \text{Ref}(\tau’, v) & \text{otherwise} \end{cases}$$

$$\text{Ref}(\tau, \{V\}) = \{\text{Ref}(\tau, \{V_1\} | V_1 ∈ \text{Ref}(\tau, V)\}$$

Table 4 shows the abstraction of virtual method invocations. The heap environment transformer for the virtual call edge is paired with a singleton set, which specifies the expected runtime type t for the call site receiver to follow this call path. Transformers for other program statements are paired with an empty set ∅ initially.

Definition 4.4 Define Ref⁻¹ : P(Expt) → P(V ∪ Obj) such that Ref⁻¹(V) = \bigcup_{t ∈ V} Ref⁻¹(t)(vt) for V ∈ Expt, where

$$\text{Ref}^{-1}([\{vt\}]) = \{vt\}$$

$$\text{Ref}^{-1}(\text{henv}(v)) = \{v\}$$

$$\text{Ref}^{-1}(\text{henv}(\text{Expt}(vt))) = \{\text{Ref}^{-1}(\text{Expt}(vt))\}$$

$$\text{Ref}^{-1}(\text{henv}(\text{Expt})) = \{\text{Ref}^{-1}(\text{Expt})\}$$

Definition 4.5 Define trace: ExpFun → V → P(V ∪ Obj) such that trace = Ref⁻¹ ◦ Ref.

Example 4.6 Let τ = λhenv. henv • [x ⇔ henv(y)] • [henv(z) ⇔ o]. Then trace(τ, x, f) = y.f, trace(τ, z, f) = o, and trace(τ, y) = y.

Definition 4.7 Let e ⊆ PathCons and τ ∈ ExpFun. For (s, t | m) ∈ e,

$$\text{judge}(e, τ) = \begin{cases} \text{error} & \text{if there exists } (s, t | m) ∈ e \text{ s.t.} \\ \text{judge}((s, t | m), τ) = \text{error} \\ \text{judge}((s, t | m), τ') & \text{otherwise} \end{cases}$$

\bigcup_{(s,t|m)∈e} \text{judge}((s, t | m), τ)
their start time. To reason about this bi-dimensional model, two types of temporal operators are applied: local and global ones. Local operators apply to the sequence of states inside the session, for instance, the "previously local" operator \( Y_L \) refers to the previous state in the same session, while "previously global" \( \{ Y_L \} \) points to the final state of the previous session.

### 3 Experience and Discussion

The S3MS.NET run time monitor was developed in the European FPs project, Security of Software and Services for Mobile Systems (S3MS). The tool is a component of a comprehensive security architecture for mobile devices [2] that supports a novel paradigm for developing trustworthy applications, the security-by-contract paradigm [3].

The implementation of the tool, as well as supporting documentation and examples can be found at http://www.cs.kuleuven.be/~pieter/inliner/

Space limitations make it impossible to discuss related research in this paper. We refer to the public S3MS deliverables at http://www.s3ms.org for a detailed overview of related work. Here, we limit ourselves to a brief summary of our experiences with the tool.

In the context of the S3MS project, we gained experience with the tool described in this paper on two case studies:

- a “Chess-by-SMS” application, where two players can play a game of chess on their mobile phones over SMS.
- a multiplayer online role-playing game where many players can interact in a virtual world through their mobile phones. The client for this application is a graphical interface to this virtual world, and the server implements the virtual world, and synchronizes the different players.

In addition, the tool was used to support a project assignment for a course on secure software development at the K.U.Leuven. In this assignment, students were asked to enforce various policies on a .NET e-mail client.

Based on these experiences, we summarize the major advantages and limitations of the tool.

A major advantage of the tool, compared to state-of-the-art code access security systems based on sandboxing (such as .NET CAS and the Java Security Architecture) is its improved expressiveness. The main difference between CAS and the approach outlined here, is that CAS is stateless. This means that in a CAS policy, a method call is either allowed for an application or disallowed. With the S3MS approach, a more dynamic policy can be written, where a method can for instance be invoked only a particular number of times. This is essential for enforcing policies that specify quota on resource accesses.

A second important strength of the tool is its performance. A key difference between CAS and our approach, is that CAS performs a stack walk whenever it tries to determine whether the application may invoke a specific sensitive function or not. Because stack walks are slow, this may be an issue on mobile devices (CAS is not yet implemented on the .NET Compact Framework). The speed of the S3MS

For this purpose, the basic weight functions, i.e., the abstract heap environment transformers (Definition 4.1), are extended by pairing path constraints (Definition 4.2). We denote by \((s, t \vdash m)\) a path constraint \((s, t, m) \in \text{PathCons}\), which intends that the dynamic dispatch of a call edge demands the runtime type of the heap object pointed to by \(s\) to be compatible with the type \(t\) w.r.t. the method \(m\). This judgement on types should exactly obey to (such as Definition 3.5) or soundly approximates the Java semantics for dynamic dispatch.

### 4 Points-to Analysis as WPDMC

#### 4.1 The Design of Weight Space

By encoding the program as a pushdown system, we are provided with context-sensitivity regarding valid paths. To construct a context-sensitive call graph during the analysis, we enrich the notion of valid paths, such that valid paths that violate type requirements of dynamic dispatch are also regraded as invalid. By encoding dataflow as weights, an invalid control flow is detected as that carrying conflicting dataflow, and combining weights along the control flow will result in the weight 0.

#### Definition 4.1 Define abstract heap environment transformers ExpFun as,

For this purpose, the basic weight functions, i.e., the abstract heap environment transformers (Definition 4.1), are extended by pairing path constraints (Definition 4.2). We denote by \((s, t \vdash m)\) a path constraint \((s, t, m) \in \text{PathCons}\), which intends that the dynamic dispatch of a call edge demands the runtime type of the heap object pointed to by \(s\) to be compatible with the type \(t\) w.r.t. the method \(m\). This judgement on types should exactly obey to (such as Definition 3.5) or soundly approximates the Java semantics for dynamic dispatch.

#### Definition 4.2 Define a set of path constraints PathCons ⊆ \( V \times T \times M \), where \( V := v \mid V, f \mid [V] \) is the set of references that syntactically allows nested field references and array structures.
approach mainly depends on the speed of the before and after handlers. These can be made arbitrarily complex, but are usually only a few simple calculations. This results in a small performance overhead. Microbenchmarks [5] show that the performance impact of the inlining itself is negligible, and for the cases and cases studies done in SIMS, there was no noticeable impact on performance.

Finally, the support for multiple policy languages and multiple platforms makes the tool a very versatile security enforcement tool.

A limitation is that we do not support applications that use reflection. Using the reflection API functions can be called dynamically at runtime. Hence, for security reasons, access to the reflection API should be forbidden, or the entire system becomes vulnerable. We do not see this as a major disadvantage, however, because our approach is aimed at mobile devices, and the reflection API is not implemented on the .NET Compact Framework. Also, by providing suitable policies for invoking the Reflection API limited support for reflection could be provided.

A second limitation of the approach implemented in the tool is that it is hard and sometimes even impossible to express certain useful policies as security automata over API method calls. For instance, a policy that limits the number of bytes transmitted over a network needs to monitor all API method calls that could lead to network traffic, and should be able to predict how much bytes of traffic the method will consume. In the presence of DNS lookups, redirects and so forth, this can be very hard.

A final limitation is that the policy languages supported by the tool are targeted to “expert” users. Writing a correct policy is much like a programming task. However, more user-friendly (e.g. graphical) policy languages could be compiled to Conspec or 2DLTL.

### References

---

**Table 2:**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Heap Environment Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = new T</td>
<td>τ = λhenv. $\text{henv} \odot {x \mapsto \nu(henv, \ldots)}$</td>
</tr>
<tr>
<td>y = y</td>
<td>τ = λhenv. $\text{henv}(y)$</td>
</tr>
<tr>
<td>x = p[]</td>
<td>τ = λhenv. $\text{henv}(p[])$</td>
</tr>
<tr>
<td>x[] = y</td>
<td>τ = λhenv. $\text{henv}(x) \odot {y}$</td>
</tr>
<tr>
<td>y = f</td>
<td>τ = λhenv. $\text{henv}(y)$</td>
</tr>
<tr>
<td>y = x</td>
<td>τ = λhenv. $\text{henv}(x)$</td>
</tr>
<tr>
<td>x = return</td>
<td>τ = λhenv. $\text{henv}(x)$</td>
</tr>
<tr>
<td>m(x, y, ...) = r</td>
<td>τ = $\nu_0 \ldots \nu_n \in \text{Pan}$</td>
</tr>
</tbody>
</table>

where $\nu_0, \ldots, \nu_n \in \text{Pan}$.

Pan = \{ (henv, $\arg_0 \mapsto \text{henv}(m_0)$) | ...

The method $m$ to be invoked is from the class with which the type of $x(y)$ is compatible with respect to the method $m$.

(Definition 3.5, i.e., the basic procedure of dynamic dispatch)

---

with any, for each $t$ in $T$ and vice versa; none conflicts with $t$, for each $t$ in $T$ and vice versa.

### 3.2 Abstraction

There are varieties of infinities to be abstracted away for a tractable analysis, such as the nesting of array structures, method invocations, field references, and the number of allocated heap objects. We take the following abstractions in the analysis,

- An unique abstract heap object models objects allocated at each heap allocation site, and is identified by its type and program line number (Definition 3.6).

Therefore, the number of abstract heap objects are syntactically bounded;

- The indices of arrays are ignored, such that members of an array are not distinguished. We denote by $[v]$ the representative for array references $v[i]$. References with nested $[\ldots]$ refer to multi-array access. We denote by $[v_0 \ldots v_k]$ by $[\ldots]$

Note that, after abstracting heap objects to be a finite set, the nesting of either field references or array references are correspondingly finite yet unbounded. Since local variables have a unique counterpart representation in the analysis, we will reuse notations in Table 1 afterwards when it is clear from the context.

**Definition 3.6** Define abstract heap objects $\Omega_{UB} = \{[t], t \in T, t \in \text{Loc} \cup \{\text{Tobj}, \ldots\}\}$, where $\text{Tobj}$ is the greatest element and $\text{Tobj} \cup \{\text{Tobj}\}$ is the least element. Other elements in $\Omega_{UB}$ except $\text{Tobj}$ and $\text{Tobj}$ are incomparable.

**Definition 3.7** An abstract heap environment $\text{henv}$ is a mapping from $\text{V}_{\text{data}}$ to $\mathcal{P}(\Omega_{UB})$, where $\mathcal{P}$ is the powerset operator. The set of abstract heap environments is denoted by $\text{Henv}_{UB}$. The evaluation function $\text{eval}_{UB}$: $\text{Henv}_{UB} \times \mathcal{V}_{\text{data}} \rightarrow \mathcal{P}(\Omega_{UB})$ on reference variables in the abstract domain is defined as:

$\text{eval}_{UB}(\text{henv}, v) = \text{henv}(v)$

$\text{eval}_{UB}(\text{henv}, [\ldots]) = \text{henv}(\ldots)$ if $v \in \text{eval}_{UB}(\text{henv}, v)$.

$\text{eval}_{UB}(\text{henv}, [\ldots]) = \text{henv}(\ldots)$ if $v \in \text{eval}_{UB}(\text{henv}, v)$.
local variables, since they can be syntactically identified as well and we limit our focus to single-thread Java programs in the presentation.

**Definition 3.2** A heap environment \( henv \) is a mapping from \( V_{	ext{ref}} \) to \( O \). The set of heap environments is denoted by \( \text{Heap}_{\text{env}} \). The evaluation function \( \text{eval}_{\text{con}} : \text{Heap}_{\text{env}} \rightarrow \text{Ranges} \) on reference variables is defined as:

\[
\text{eval}_{\text{con}}(henv, v) = henv(v)
\]

\[
\text{eval}_{\text{con}}(henv, e[i]) = henv[henv(v)[i]]
\]

\[
\text{eval}_{\text{con}}(henv, v.f) = henv(henv(v).f)
\]

Let \( h_{\text{init}} \) be the initial heap environment such that, for each \( r \in V_{\text{ref}}, \)

\[
\text{eval}_{\text{con}}(h_{\text{init}}, r) = \perp \quad \text{(null reference)}.
\]

Let \( \text{Loc} \) be the set of program locations. Since we only consider single thread Java program here, the next program location at each execution step is uniquely determined. We informally refer it as \( \text{next}(l) \) for \( l \in \text{Loc} \). Later it will be discussed how to leverage the analysis to a flow-insensitive counterpart regarding concurrency.

**Definition 3.3** Define an transition system \( \mathcal{OS} = (\text{States}, \mathcal{s}_{\text{init}}, \rightarrow) \) to represent the Java semantics on heap, where:

- \( \text{States} \subseteq (\text{Loc} \times \text{Henv}_{\text{con}}) \) is a set of pairs of a program location and a heap environment.
- \( \mathcal{s}_{\text{init}} \) is the initial state, which is a pair of the program entry \( \ell_0 \) and \( h_{\text{init}} \):
- \( \rightarrow \subseteq \text{States} \times \text{States} \) is the set of operational semantic rules, and \( \rightarrow \) denotes the transitive closure of \( \rightarrow \).

A transition rule \((l, henv) \rightarrow (\text{next}(l), \tau(henv))\) for typical pointer assignment statements at \( l \in \text{Loc} \) is shown in Table 2, where:

- the function \( c(henv, T) \) generates a fresh heap object of type \( T \) in \( \text{Con} \);
- for \( r, r' \in V_{\text{ref}}, \sigma \in \text{Con} \),

\[
(henv \odot \rho[r \mapsto o])p' = \begin{cases} o & \text{if } r = r' \\ henv(r) & \text{otherwise} \end{cases}
\]

**Definition 3.4** The composition of heap environment transformers is defined by the standard \( \eta \)-expansion, such that, for \( \text{expl}_1, \text{expl}_2 \in \text{Exp}_{\text{Henv}} \),

\[
(\text{ expl}_2 \circ (\text{ expl}_1)) = \eta \text{Henv}(\text{ expl}_2 \circ (\text{ expl}_1)) = \text{Henv}(\text{ expl}_2) \text{Henv}(\text{ expl}_1)
\]

The notation \( E[h := E'] \) means the expression \( E \) with \( E' \) substituted for free occurrences of \( h \).

**Definition 3.5** Let \( M \) be the set of method identifiers, and let \( T \) be the set of reference types. Let \( \text{any} = \text{type}(\perp) \) and \( \text{none} = \text{type}(\perp) \). For \( t, t' \in T \setminus \{ \text{any}, \text{none} \} \) and \( m \in M \), \( t', t \in M \) conflict with \( t \) with respect to the method \( m \) if and only if, (i) \( t' \neq t \) and \( t' \) does not inherit \( m \), or (ii) \( t' \) inherits \( t \) with redefining \( m \). Otherwise, we say \( t' \) is compatible with \( t \) with respect to the method \( m \). Furthermore, \( t \) is compatible with \( t' \) with respect to the method \( m \).

---

**Abstract**

Although in most cases class initialization works as expected, some static fields may be read before being initialized, despite being initialized in their corresponding class initializer. We propose an analysis which aims at generalizing the problem at hand and provide a crisp handle on the soundness. We show that such an analysis can be directly applied to identify the static fields that may be read before being initialized and to improve the precision while maintaining the soundness of a null-pointer analysis.

**Keywords:** static analysis, Java, semantics, class initialization, control flow, verification.

---

1 Introduction

Program analyses often rely on the data manipulated by programs and can therefore depend on their static fields. Unlike instance fields, static fields are unique to each class and one would like to benefit from this uniqueness to infer precise information about their content.

When reading a variable, be it a local variable or a field, being sure it has been initialized beforehand is a nice property. Although the Java bytecode ensures this property for local variables, it is not ensured for static and instance fields which have default values.

Instance fields and static fields are not initialized the same way: instance fields are usually initialized in a constructor which is explicitly called whereas static fields are initialized in class initializers which are implicitly and lazily invoked. This makes the control flow graph much less intuitive.

The contributions of this work are the following.

---

1 Eric Forest, last@irisa.fr
2 This work was supported in part by the Région Bretagne

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

---

Bytecode 2009 53 ETAPS 2009, York, UK
We recall that implicit lazy static field initialization make the control flow graph hard compute.

We identify some code examples that would need to be ruled out and some other examples that would need not to be ruled out.

We propose a language to study the initialization of static fields.

We propose a formal analysis to infer an under-approximation of the set of static fields that have already been initialized for each program point.

We propose two possible applications for this analysis: a direct application is to identify potential bugs and another one is to improve the precision while keeping the correctness of a mill-pointer analysis.

The rest of this paper is organized as follows. We recall in Sect. 2 that the actual control flow graph which includes the calls that the class initializers it not intuitive and gives some examples. We present in Sect. 3 the syntax and semantics of the language we have chosen to formalize our analysis. Section 4 then presents the analysis, first giving an informal description and then its formal definition. We then explain in Sect. 5 how the analysis can be extended to handle other features of the Java bytecode language. In Sect. 6, we give two possible applications of this analysis. Finally, we discuss the related work in Sect. 7 and conclude in Sect. 8.

2 Why Static Analysis of Static Fields is Difficult?

The analysis we herein present works at the bytecode level but, for sake of simplicity, code examples are given in Java. As this paper is focused on static fields, all fields are assumed to be static unless otherwise stated.

In Java, a field declaration may include the initial value, such as k.f in Fig. 1. A field can also be initialized in a special method called a class initializer, which is identified in the Java source code with the static keyword followed by no signature and a method body, such as in class B in the same figure. If a field is initialized with a compile-time constant expression, the compiler from Java to bytecode may translate the initialization into a field initializer (cf. [12], Sect. 4.7.2), which is an attribute of the field. At run time, the field should be set to this value before running the class initializer. In-line initializations that have not been compiled as field initializers are prepended in textual order to the class initializer, named <clinit> at the bytecode level. For this analysis, we do not consider field initializers but focus on class initializers as they introduce the main challenges. Although this simplification is sound, it is less precise and we explain how to extend our analysis to handle field initializers in Sect. 5.3.

The class initialization process is not explicitly handled by the user: it is forbidden to explicitly called a <clinit> method. Instead, every access (read or write) to a field of a particular class or the creation of an instance of that same class requires that the JVM (Java Virtual Machine) have invoked the class initializer of that class. This method can contain arbitrary code and may trigger the initialization of other classes and so on.

The JVM specification [12] requires class initializers to be invoked lazily. This implies that the order in which classes are initialized depends on the execution path.

### Table 1

<table>
<thead>
<tr>
<th>f</th>
<th>∈</th>
<th>F</th>
<th>Name constants of instance fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>∈</td>
<td>RefVar</td>
<td>Local variables and static fields</td>
</tr>
<tr>
<td>v[i]</td>
<td>∈</td>
<td>RefArr</td>
<td>Array references (i ∈ Z^0_)</td>
</tr>
<tr>
<td>v[i]</td>
<td>∈</td>
<td>Vref</td>
<td>RefVar ∪ RefArr ∪ RefVar × F</td>
</tr>
<tr>
<td>v[i]</td>
<td>∈</td>
<td>Vdef</td>
<td>RefVar ∪ O × Z^0_ ∪ O × F</td>
</tr>
</tbody>
</table>

Recall the usual encoding of programs as finite model checking, program states, i.e., the product of global variables, local variables and program execution points, are encoded as states of finite automatons. For pushdown model checking, the pushdown stack can simulate the runtime stack of program execution. For instance, the pushdown stack can be encoded to store calling contexts for procedure calls, just like the program execution on stack machine. In this paper, we will follow the convention defined in Definition 2.6.

**Definition 2.6** Define an interprocedural control flow graph \( G = (N, E, v_0) \), where \( N = N_1 ∪ N_2 ∪ N_3 \) is the set of nodes, with \( N_1, N_2, N_3 \) as the sets of internal nodes, call sites, and method exits, respectively. \( E = E_0 ∪ E_1 ∪ E_2 \) is the set of edges with \( E_0 ⊆ N_1 × N_2, E_1 ⊆ N_1 × N_3, E_3 ⊆ N_2 × N_3 \) where \( E_0, E_1, \) and \( E_3 \) are the sets of internal edges, call edges, and return edges, respectively. \( v_0 ∈ N \) is the unique entry node of \( G \). We denote by \( N_i ⊆ N \) the set of return points of method calls. Let \( \text{assign} : N_i → N \) be the function that associates with each call site from \( N_i \) with a distinguished return point in \( N \), \( N_i = \{ n_i | n_i = \text{assign}(n) \in N \} \).

**Definition 2.7** The encoding of an interprocedural control flow graph \( G = (N, E, v_0) \) as a pushdown system \( P = (Q, Γ, Δ, q_0, v_0) \) is defined as follows:

*\( Q \) is a singleton set denoted by \( \{ \} \);

*\( Γ = N \) with \( v_0 = v_0 \);

*\( Δ \) is constructed as follows,

\[
\begin{align*}
(\cdot, n_i) & \rightarrow (\cdot, n'_i) \neven (n_i, n'_i) \in E_2 \\
(\cdot, n_i) & \rightarrow (\cdot, n_i n) \neven (n_i, n_i) \in E_3 \text{ and } n_i = \text{assign}(n_i) \in N_i \\
(\cdot, n) & \rightarrow (\cdot, \epsilon) \neven (n_0, n) \in E_0 
\end{align*}
\]

3 Modelling and Abstraction

3.1 Semantics of Heap Operations

**Definition 3.1** Define \( O \) be the set of heap objects in the concrete domain, where \( T_o ∈ O \) is the greatest element and represents any objects; \( ⊥_o ∈ O \) is the least element and represents no objects (i.e., null reference). Elements in \( O \) except \( T_o \) and \( ⊥_o \) are incomparable.

We take Jimple, a three-address intermediate representation of Java, as our target language, since it is syntactically much simpler than either Java or Bytecode. Table 1 prepares notations for, (i) the set of references \( V_{ref} \) that is syntactically allowed in Jimple; and (ii) the set of references \( V_{def} \) in the semantic domain of heap environments (Definition 3.2).

Static fields are treated in the same way with
class A extends Object {
    static B f = new B();
}

class B extends Object {
    static B g;
    static {
        g = A.f;
    }
}

Fig. 1. Initial values can depend on foreign code: in this example, the main program should first use B for the initialization to start from B to avoid B.g to be null.

As stated in Section 4.4 in [8], the distributivity of $\otimes$ can be loosened to $a \otimes (b \oplus c) \subseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \subseteq (a \otimes c) \oplus (b \otimes c)$. The associativity of $\otimes$ can be loosened too, as long as both $(a \otimes b) \otimes c$ and $a \otimes (b \otimes c)$ conservatively approximates the program execution when applied to program analysis.

2.5 Consider a weighted pushdown system $W = (P, S, f, \sigma)$ where $P = (Q, \Gamma, \Delta, \phi, \psi)$ is a pushdown system, $S = (D, \oplus, \ominus, 0, 1)$ is a bounded idempotent semiring, and $f : \Delta \to D$ is a function that assigns a value from D to each rule of P.

The GPR can be easily extended to answer the classic “meet-over-all-valid-paths” problem in program analysis. Efficient algorithms for solving GPR are developed based on the property that the regular set of pushdown configurations is closed under forward and backward reachability [8]. There are two off-the-shelf implementations of weighted pushdown model checking algorithms, Weighted PDS Library 5, and WPDS+ 6. We apply the former as the back-end analysis engine in the prototype implementation.

2.2 Program Analysis as WPDMC

When designing a program analysis as WPDMC, the intuition behind $\otimes$ and $\oplus$ is:

- A weight function models a transfer function which typically represents the data flow changes for one-step program execution;
- $f \otimes g$ represents the merging of data flow at the meet of two control flows;
- $f \oplus g$ represents the sequential composition of abstract state transformers;
- 1 implies that an execution step does not change the program state; and
- 0 implies that the program execution is interrupted by an error.

\[ \delta(c, C) = \bigoplus \{ e(\sigma) | \sigma \in \text{path}(c, c'), c' \in C \} \]

for $c \in Q \times \Gamma^*$ and a set $C \subseteq Q \times \Gamma^*$ of regular configurations.

The JVM specification also requires class initializers to be invoked at most once. This avoids infinite recursions in the case of circular dependencies between classes, but it also implies that when reading a field it may not contain yet its “initial” value. For example, in Fig. 1, the class initializer of A creates an instance of B and therefore requires that the class initializer of B has been invoked. The class initializer of B reads a field of A and therefore requires that the class initializer of A has been invoked.

- If A.<clinit> is invoked before A.<clinit>, then the read access to the field A.f triggers the invocation of A.<clinit>. Then, as B.<clinit> has already been invoked, A.<clinit> carries on normally and creates an instance of class B, store its reference to the field A.f and returns. Back in B.<init>, the field A.f is read and the reference to the new object is also affected to B.g.
- If A.<clinit> is invoked before B.<clinit>, then before allocating a new instance of B, the JVM has to initialize the class B by calling B.<clinit>. In B.<clinit>, the read access to A.f does not trigger the initializer of A because A.<clinit> has already been started. B.<clinit> then reads A.f, which has not been initialized yet, B.g is therefore set to the default value of A.f which is the null constant.

This example shows that the order in which classes are initialized modifies the semantics. The issue shown in Fig. 1 is not limited to reference fields. In the example in Fig. 2, depending on the initialization order, A.CST will be either 0 or 5, while B.SIZE will always be 5.

One could notice that those problems are related to the notion of circular dependencies between classes and may think that circular dependencies should be avoided. Figure 3 shows an example with a single class. In (a), A.ALL is read in the constructor before it has been initialized and it leads to a NullPointerException.
3 The Language

In this section we present the program model we consider in this work. This model is a high level description of the bytecode program that discards the features that are not relevant to the specific problem of static field initialization.

3.1 Syntax

We assume a set $P$ of program points, a set $F$ of field names, a set $C$ of class names and a set $M$ of method names. For each method $m$, we note $m.first$ the first program point of the method $m$. For convenience, we associate to each method a distinct program point $m.last$ which models the output point of the method. For each class $C$ we note $C<\text{clinit}>$ the name of the class initializer of $C$. We only consider four kinds of instructions.

- `put(f)` updates a field $f \in F$.
- `invoke` calls a method (we do not mention the name of the target method because it would be redundant with the `flowlatest` information described below).
- `return` return from a method.
- `any` models any other intra-procedural instruction that does not affect static fields.

As the semantics presented below will demonstrate, any instruction of the standard sequential Java bytecode can be represented by one of these instructions.

![Fig. 1. Points-to Analysis on Jimple in the Soot Framework](image)

been originally designed to simplify the process of developing new optimizations for Java bytecode and supports three kinds of intermediate representations of bytecode.

Definitions

Definition 2.1 A pushdown system $P = (Q, \Gamma, \Delta, q_0, \omega_0)$ is a pushdown automaton regardless of input, where $Q$ is a finite set of states called control locations, and $\Gamma$ is a finite set of stack alphabet, and $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$ is a finite set of transition rules, and $q_0 \in Q$ and $\omega_0 \in \Gamma^*$ are the initial control location and stack contents respectively. We denote the transition rule $(\langle q_1, w_1 \rangle, (q_2, w_2)) \in \Delta$ by $\langle q_1, w_1 \rangle \rightarrow (q_2, w_2)$. A configuration of $P$ is a pair $\langle q, \omega \rangle$, where $q \in Q$ and $\omega \in \Gamma^*$. $\Delta$ defines the transition relation $\rightarrow$ between pushdown configurations such that if $\langle p, \gamma, \omega \rangle \rightarrow \langle q, \omega' \rangle$, for all $\omega' \in \Gamma^*$. A pushdown system is a finite transition system carrying an unbounded stack. A weighted pushdown system extends a pushdown system by associating a weight to each transition rule. The weights come from a bounded idempotent semiring.

Definition 2.2 A bounded idempotent semiring $S = (D, \oplus, \odot, 0, 1)$ consists of a set $D (0, 1 \in D)$ and two binary operations $\oplus$ and $\odot$ on $D$ such that

(i) $(D, \oplus)$ is a commutative monoid with $0$ as the unit element, and $\oplus$ is idempotent, i.e., $a \oplus a = a$ for $a \in D$;
(ii) $(D, \odot)$ is a monoid with $1$ as the unit element;
depend on points-to analysis to reason the underlying control/data flow of Java programs. Due to dynamic object-oriented features like dynamic dispatch \(^5\), points-to analysis is mutually dependent to call graph construction. Thus we have choices of constructing call graph either on-the-fly as the points-to sets of call site receivers are computed, or ahead-of-time based on syntactical information of the program such as CHA \([21]\) and RTA (Rapid Type Analysis) \([22]\). The former essentially enjoys a higher precision and is the choice of most of points-to analysis algorithms.

This paper presents an ahead-of-time yet context-sensitive points-to analysis for Java as all-in-one WPDMC (weighted pushdown model checking). Though it is well understood that program analysis can be regarded as model checking of abstract interpretation \([1]\), model checking based approach to a context-sensitive points-to analysis is not straightforward. We limit our focus to the following context-sensitivities in the analysis, such that (i) method calls and returns match with each other (a.k.a., valid paths), which is guaranteed by encoding the program as a pushdown system; and (ii) targets of dynamic dispatch are analyzed separately for different calling contexts (a.k.a., context-sensitive call graph). Our approach to (ii) is, by further encoding dataflow as weights, invalid control flows that violates Java semantics on dynamic dispatch are detected as those carrying conflicted dataflow. These context-sensitivities are recently shown to be crucial to a precise points-to analysis in practice \([5,18]\), as illustrated by Example 1.1. Our analysis is also flow-sensitive and field-sensitive. Concerning efficiency and concurrent behaviors of Java programs, points-to analysis is typically designed as flow-insensitive. One smart idea is combining SSA (Static Single Assignment) and complete flow insensitivity \([7]\). We briefly discussed how to easily obtain flow-insensitivity as a hierarchy.

**Example 1.1** We denote by \(\varphi\) an abstract heap object that is allocated at the program line \(l\), and by \(\mapsto\) the mapping relation afterwards. An analysis will precisely compute \(\{c \mapsto \varphi, d \mapsto \varphi'\}\) if it obeys to valid paths, and will furthermore erroneously infer \(\{c \mapsto \varphi, d \mapsto \varphi''\}\) otherwise. An analysis will precisely compute \(\{\varphi_1 \mapsto \varphi_1, \varphi_2 \mapsto \varphi_2\}\) if a context-sensitive call graph is constructed, and will furthermore erroneously infer \(\{\varphi_3 \mapsto \varphi_3, \varphi_4 \mapsto \varphi_4\}\) otherwise.

```
1. public class Main { 12. public class A { 2. 2. public static void main(String[] args) { 13. Object f; 3. 3. A a = new A(); 14. public void set() { 4. 4. C = foo(a); 15. this.f = new int[0]; 5. 5. A b = new B(); 16. } 6. 6. A d = foo(b); 17. } 7. } 18. public class B extends A { 8. 8. public static A foo(A x) { 19. public void set() { 9. 9. x.set(); 20. this.f = new String(); 10. return x; } 21. } 11. } 22. } 13. Object f; 23. }
```

Due to the lack of control flow structure and explicit operand stack-based design, static analysis on bytecode is not an easy matter. We thus design and implement the analysis as a sub-phase of the compilation procedure in the Soot framework. Soot is an open-source compilation/optimization framework for Java, which has

\(^{5}\) In this paper, we limit our focus to simple dynamic dispatch only. Multiple dynamic dispatch, e.g., reflection in Java, demands non-trivial extension and thus independent discussion.

The program model we consider is based on control flow relations that must have been computed by some standard control flow analysis.

**Definition 3.1** A program is a 5-tuple \((m_0, \text{instr}, \text{flow}_{\text{intra}}, \text{flow}_{\text{inter}}, \text{flow}_{\text{call}})\) where:

- \(m_0 \in M\) is the method where the program execution starts;
- \(\text{instr} \in P \rightarrow \{\text{part}, \text{invoke}, \text{return}, \text{any}\}\) is a partial function that associates program points to instructions;
- \(\text{flow}_{\text{intra}} \subseteq P \times P\) is the set of intra-procedural edges;
- \(\text{flow}_{\text{inter}} \subseteq P \times M\) is the set of inter-procedural edges, which can capture dynamic method calls;
- \(\text{flow}_{\text{call}} \subseteq P \rightarrow C\) is the set of initialization edges which forms a partial function since an instruction may only refer to one class;

and such that \(\text{instr}\) and \(\text{flow}_{\text{intra}}\) satisfy the following property:

For any method \(m\), for any program point \(l \in P\) that is reachable from \(m\) first in the intra-procedural graph given by \(\text{flow}_{\text{intra}}\) and such that \(\text{instr}(l) = \text{return}, (l, m, last)\) belongs to \(\text{flow}_{\text{call}}\).

In practice, \(\text{flow}_{\text{call}}\) will contain all the pairs \((l, C)\) of a bytecode program such that the instruction found at program point \(l\) is of the form \(\text{new } C, \text{ putstatic } C.f, \text{ getstatic } C.f\) or \(\text{invokestatic } C.m\) (see Section 2.17.4 of \([12]\)).

Figure 7 in Sect. 4.2 presents an example of program with its three control flow relations. In this program, the main method \(m_0\) contains two distinct paths that lead to the call of a method \(m\). In the first one, the class \(A\) is initialized first and its initializer triggers the initialization of \(B\). In the second path, \(B\) is not initialized but \(B.f()\) is potentially called at exit point 8 but since 8 is only reachable after a first call to \(B<\text{clinit}\), this initialization edge is never taken.

### 3.2 Semantics

The analysis we consider does not take into account the content of heaps, local variables or operand stacks. To simplify the presentation, we hence choose an abstract semantics which is a conservative abstraction of the concrete standard semantics and does not explicitly manipulate these domains.

The abstract domains the semantics manipulates are presented below.

- \(v \in \text{Value} (\text{abstract})\)
- \(s \in \text{Static} \triangleleft F \rightarrow \text{Value} + \{\Omega\}\)
- \(h \in \text{History} \triangleleft F(C)\)
- \(c.s \in \text{Callstack} \triangleleft \{(0,1) \times F\}\)
- \(\langle l, c, s, h \rangle \in \text{State} \triangleleft F \times \text{Callstack} \times \text{Static} \times \text{History}\)

To be completely correct we also need to add an edge from the beginning of the static void main method to the initializer of its class. We can also handle correctly superclass and interface initialization without deep modification of the current formalization.
An Ahead-of-time Yet Context-Sensitive Points-to Analysis for Java

Xin Li1,2
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Japan

Mizuhito Ogawa3
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Japan

Abstract
Points-to analysis is a prerequisite of program verification and static analysis on Java programs. It is known that call graph is typically constructed on-the-fly when points-to analysis proceeds for a better precision. In this work, we propose an ahead-of-time yet context-sensitive points-to analysis for Java as an in-line weighted peakdown model checking. The analysis is context-sensitive in the sense that (i) methods calls and returns match with each other (a.k.a., solid paths); and (ii) targets of dynamic dispatch are analyzed separately for different calling contexts (a.k.a., context-sensitive call graph). The insight of our approach separates dispatch into two phases: (i) dispatch analysis is performed separately, and (ii) the hand-coded dispatch are detected as those carrying conflicted dataflow. An analysis is presented as field-sensitive and flow-sensitive. Flow-missensitivity is shown to be easily obtained as a byproduct considering efficiency and concurrent behavior. Due to the lack of control flow structure and the explicit stack-based design, program analysis on bytecode is not an easy matter. We implemented the analysis in the framework of Soot compiler, and utilized the Weighted PDS Library as the back-end analysis engine. The analysis works on Jimple, a typed three-address intermediate representation of bytecode supported by Soot. The results of the analysis can be encoded into the class file as attributes for the further analysis or verification on bytecode.

Keywords: Point-to Analysis, Weighted Peakdown Model Checking, Java

1 Introduction
Points-to analysis [4] for Java is to detect the set of heap objects, i.e., instances of classes or arrays, possibly referred to by reference variables at run-time. Many applications such as program understanding, program verification, and static analysis

1 We would like to thank anonymous reviewers for their valuable and thorough comments. This research is supported by the 21st Century COE program "Verifiable and Exchanging e-Society" of JAIST, funded by the Japanese Ministry of Education, Culture, Sports, Science, and Technology.
2 Email: li-xin@jaist.ac.jp
3 Email: mizuhito@jaist.ac.jp

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

Fig. 4. Operational semantics
control is transferred to (one of) the target(s) of the inter-procedural edge. At last, the instruction return requires two rules. In the first case, a standard method call, the transfer comes back to the intra-procedural successor of the caller. In the second case, a class initializer call, we have finished the initialization and we must now use the standard semantic \( \rightarrow_1 \) of the pending instruction in program point \( l \).

We end this section with the formal definition of the set of reachable states during a program execution. An execution starts in the main method \( m_0 \) with an empty call stack, an empty historic and with all fields associated to the default value \( \Omega \).

**Definition 3.2** [Reachable States] The set of reachable states of a program \( p = (m_0, \text{inst}, \text{flow}_{\text{intra}}, \text{flow}_{\text{inter}}, \text{flow}_{\text{clinit}}) \) is defined by

\[
[p] = \{ <i, cs, s, h> | <m_0, \text{first}, c, A_f, \emptyset, \emptyset> \rightarrow^* <i, cs, s, h> \}
\]

4 A Must-Have-Been-Initialized Data Flow Analysis

In this section we present a sound data flow analysis that allows to prove a static field has already been initialized at a particular program point. We first give an informal presentation of the analysis, then present its formal definition and we finish with the statement of a soundness theorem.

4.1 Informal presentation

For each program point we want to know as precisely as possible, which fields we are sure we have initialized. Since fields are generally initialized in class initializers, we need an inter-procedural analysis that infers the set of fields \( W_f \) that must have been initialized at the end of each method. Hence at each program point \( l \) where a method is called, be it a class initializer or another method, we will use this information to improve our knowledge about initialized fields.

However, in the case of a call to a class initializer, we need to be sure the class initializer will be effectively executed if we want to safely use such an information. Indeed, at execution time, when we reach a point \( l \) with an initialization edge to a class \( C \), despite \( \text{flow}_{\text{clinit}}(l) = C \), two exclusive cases may happen:

(i) \( C.<\text{clinit}> \) has not been called yet. \( C.<\text{clinit}> \) is immediately called.
(ii) \( C.<\text{clinit}> \) has already been called (and may still be in progress): \( C.<\text{clinit}> \) will not be called a second time.

Using the initialization information given by \( C.<\text{clinit}> \) is safe only in case (i), i.e. the control flow information in \( \text{flow}_{\text{clinit}} \) is not precise enough. To detect case (i), we keep track in a flow-sensitive manner of the class initializer that may have been called during all execution reaching a given program point. We denote by \( May \) this set. Here, if \( C \) is not in \( May \), we are sure to be in case (i). \( May \) is computed by gathering, in a flow sensitive way, all classes that may be initialized starting from the main method. Implicit calls to class initializer need to be taken in account, but the smaller \( May \) is, the better.
1 class A{ static int f = 1; }
2 class B{ static int g = A.f; }
3 class C{
4     public static void main ( String[] args ){
5         ... = B.g;
6         ... = A.f;
7         ... = B.g;
8     }
9 }

Fig. 5. Motivating the Must set

For the simplicity’s sake we consider in this work a context-insensitive analy-
sis where for each method, all its calling contexts are merged at its entry point.
Consider the program example given in Fig. 5. Before line 5, May only contains
C, the class of the main method. There is an implicit flow from line 5 to the class
initialization of B. At the beginning of the class initialization of B, May equals to \{B,C\}. We compute the set of fields initialized by A.<clinit>, which is \{A.f\}. As A is not in May at the beginning of B.<clinit>, we can assume the class initial-
ization will be fully executed before the actual read to A.f occurs, so it is a safe read.
However, when we carry on line 7, the May set contains A, B and C. If we flow
this information to B.<clinit> then the merged calling context of B.<clinit> is now
\{A,B,C\} since \{A.f\} is propagated at line 2. To avoid such an imprecision, we try to propagate as few calling context as possible to class initializers by computing in a flow-sensitive manner a second set of class whose initializer must have already been called in all execution reaching a given program point. We denote by Must this set. Each time we encounter an initialization edge for a class C, we add C to Must since C.<clinit> is either called at this point, or has already been called before. If C ∈ Must before an initialization edge for C, we are sure C.<clinit> will not be called at this point and we can avoid to propagate a useless calling context to C.<clinit>.

To sum up, our analysis manipulates, in a flow sensitive manner, three sets May, Must and WF. May and Must correspond to a control flow analysis that allows to refine the initialization graph given by flow.<clinit>. The more precise control flow graph allows a finer tracking of field initialization and therefore a more precise WF.

### 4.2 Formal specification

In this part we consider a given program \( p = (m_a, instr, flow_{static}, flow_{inst}, flow_{clinit}) \). We note \( P_l(C) \) (resp. \( P_l(F) \)) the finite set of classes (resp. fields) that appears in \( p \). For each program point \( l \in P \), we compute before and after the current point three sets of data (May, Must, WF) \( \in P_l(C) \times P_l(C) \times P_l(F) \). Since May is a may information, and Must and WF are must information, the underlying lattice of the data flow analysis is given by the following definition.

**Definition 4.1 [Analysis lattice]** The analysis lattice is \( (A^I, \sqsubseteq, \sqcup, \sqcap, \bot, \top) \) where:

- \( A^I = P_l(C) \times P_l(C) \times P_l(F) \).
inference from propagating. Annotations are also used by the size and resource usage analysis to express their output. Finally, the annotation language can also be used to state specifications related to resource usage, which can then be proved or disproved based on the results of analysis following, e.g., the scheme of [24, 5, 22] thus finding resource bugs or verifying the resource usage of the program.

**References**


\[ A_{in}(l) = A_{0}(l) \sqcup A_{first}(l) \sqcup \bigcup \{ A_{out}(l') \mid \text{flow}_{\text{intra}}(l', l) \} \]
\[ A_{out}(l) = \begin{cases} 
F_{\text{call}}(F_{\text{init}}(l, A_{in}(l)), \bigcup \{ A_{in}(m.\text{last}) \mid \text{flow}_{\text{intra}}(l, m) \}) & \text{if instr}(l) = \text{invoke} \\
\bot & \text{otherwise}
\end{cases} \]

where
\[ A_{0}(l) = \begin{cases} 
(\emptyset, \emptyset, \emptyset) & \text{if } l = m.\text{first} \\
\bot & \text{otherwise}
\end{cases} \]
\[ A_{first}(l) = \begin{cases} 
\bigcup \{ F_{\text{call}}(C, A_{in}(l')) \mid \text{flow}_{\text{intra}}(l', l) = C \} & \text{if } l = C.\text{<clinit>}, \text{first} \\
\bot & \text{otherwise}
\end{cases} \]

and \( F_{\text{call}}, F_{\text{exec}}, F_{\text{inst}}(l, A_{in}(l)) \in A_{1} \rightarrow A_{1}, F_{\text{call}} \in A_{1} \times A_{1} \rightarrow A_{1}, F_{\text{exec}} \in C \times A_{1} \rightarrow A_{1}, F_{\text{inst}} \in A_{1} \rightarrow A_{1} \) are transfer functions defined by:
\[ F_{\text{call}}(\text{May}, \text{Must}, \text{WF}) = F_{\text{exec}}(\text{May}, \text{Must}, \text{WF}) = (\text{May}, \text{Must}, \text{WF}) \]
\[ F_{\text{inst}}(\text{May}, \text{Must}, \text{WF}) = (\text{May}, \text{Must}, \text{WF} \cup \{ f \}) \]
\[ F_{\text{call}}(\text{May}, \text{Must} \sqcup \text{WF} \sqcup \{ l \}) = (\text{May}, \text{Must} \sqcup \text{WF} \sqcup \{ f \}) \]
\[ F_{\text{inst}}(C, \text{May}, \text{Must}, \text{WF}) = \begin{cases} 
\bot & \text{if } C \in \text{Must} \\
(\text{May} \sqcup \{ C \}, \text{Must} \sqcup \{ C \}, \text{WF}) & \text{otherwise}
\end{cases} \]
\[ F_{\text{inst}}(l, a) = \begin{cases} 
F_{\text{call}}(a, A_{in}(C.\text{<clinit>}, \text{last})) & \text{if } \text{flow}_{\text{intra}}(l) = C \text{ and } C \notin \text{May} \\
a & \text{if } (\text{flow}_{\text{intra}}(l) = C \text{ and } C \in \text{Must}) \text{ or } \forall C. (\text{flow}_{\text{intra}}(l) \neq C) \\
\text{a} \sqcup F_{\text{call}}(a, A_{in}(C.\text{<clinit>}, \text{last})) & \text{otherwise}
\end{cases} \]

The equation on \( A_{out}(l) \) distinguishes two cases, depending on \( \text{instr}(l) \) is a method call or not. If it is, we merge all data flows from the end of the potentially called methods and combine them, using \( F_{\text{call}} \) described below, with the data flow facts \( \text{flow}_{\text{inst}}(l, A_{out}(l)) \) that is true just before the call. Otherwise, we transfer the data flow \( A_{out}(l) \), found at entry of the current instruction with \( F_{\text{inst}} \) and then \( F_{\text{call}} \). While \( F_{\text{inst}} \) handles potential class initialization that may have been performed before the instruction, \( F_{\text{call}} \) simply handles the effect of the instruction

### 6 Conclusions
We have presented a fully-automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. Our analysis derives a vector of functions, one for each defined resource. Each of these functions returns, for each given set of input data sizes, an upper bound on the usage that the whole program (and each individual method) make of the corresponding resource. Our approach allows the application programmer to define the resources to be tracked by writing simple resource descriptions via source-level annotations. The current results suggest that the proposed analysis can obtain non-trivial bounds on a wide range of interesting resources in reasonable time. Our approach allows using the annotations also for a number of other purposes such as stating the resource usage of external methods, which is instrumental in allowing modular composition and thus scalability. In addition, our annotations allow stating the resource usage of any method for which the automatic analysis infers a value that is not accurate enough to prevent inaccuracies in the automatic...
The problem domain is a multithreaded bytecode-based resource consumption analyzer. We have developed a tool that captures the resource usage of arbitrary Java bytecode and can be used for static resource analysis. Our tool is based on a novel dataflow analysis framework and is implemented in Ciao. We have performed an evaluation of our tool on a set of benchmarks and found that it is able to accurately approximate the resource usage of bytecode-level resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program Files (a fragment characteristic of operating system kernel

eral, approximate, but always safe) closed form upper bound on the resource usage of the block methods in each strongly-connected component. Note that given a signature the analysis constructs a closed form solution for every block method that shares that signature. These solutions approximate the resource usage consumed in or provided by each block method. In order to compute the total resource usage of the signature the analysis returns the maximum of these solutions yielding a safe global upper bound.

Example 5 The resource usage equations generated by our algorithm for the two sendSms block methods and the "$" resource (monetary cost of sending the SMSs) are listed in Fig. 7. The computation is partially based on the size relations in Fig. 5. The resource usage of each block method is calculated by building an equation such that the left part is a symbolic function constructed by replacing each parameter position with its size (i.e., Cost (f, s_mₐ₅₉, s₁, s₂, s₃)) and the rest of the equation consists of adding the resource usage of the invoke statements in the block method. These are calculated by computing the minimum between the resource usage function inferred by the analysis and the function provided by the user. The equations corresponding to the recursive and non-recursive block methods are in the first and second row, respectively. They can be simplified (third row) and expressed in closed form (fourth row), obtaining a final upper bound for the charge incurred by sending the list of text messages of $6 x s₃^2 - 6 x s₄^1$.

5 Experimental results

We have completed an implementation of our framework (in Ciao [10], using components from CiaoPP [23], and with help from the Soot tool [39], as mentioned before), and tested it for a representative set of benchmarks and resources. Our experimental results are summarized in Table 1. Column Program provides the name of the main class to be analyzed. Column Resource(s) shows the resource(s) defined and tracked. Column $t_ι$ shows the time (in milliseconds) required by the size analysis to construct the size relations (including the data dependency analysis and class hierarchy analysis) and obtain the closed form. Column $t_ο$ lists the time taken to build the resource usage expressions for all method blocks and obtain their closed form solutions. $t$ provides the total times for the whole analysis process. Finally, column Resource Usage Func. provides the upper bound functions inferred for the resource usage. For space reasons, we only show the most important (asymptotic) component of these functions, but the analysis yields concrete functions with constants.

Regarding the benchmarks we have covered a reasonable set of data-structures used in object-oriented programming and also standard Java libraries used in real applications. We have also covered an ample set of application-dependent resources which we believe can be relevant in those applications. In particular, not only do we represent high-level resources such as cost of SMS, bytes received (including a coarse measure of bandwidth, as a ratio of data per program step), and files left open, but also other low-level (i.e., bytecode level) resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program Files (a fragment characteristic of operating system kernel

Fig. 7. Program and analysis example
**Theorem 4.5 (Soundness)** If \((A_{in}, A_{out})\) is a data flow solution then for all reachable states \(\langle s, r \rangle \in [p]\). \(A_{in}(s) \sim (s, h)\) holds.

**Proof sketch.** We first define an intermediate semantics \(\sim\) which is shown equivalent to the small-step relation \(\Rightarrow\), but in which method calls are big-steps: for each point \(t\) where a method is called we go in one step to the intra-procedural successor of \(t\) using the result of the transitive closure of \(\Rightarrow\). Such a semi-big-step semantics is easier to reason with than method calls. Once \(\sim\) is defined, we prove a standard subject reduction lemma between \(\sim\) and \(\Rightarrow\) and we conclude. \(\Box\)

5 Handling the Full Bytecode  

5.1 Exceptions

From the point of view of this analysis, exceptions only change the control flow graph. As the control flow graph is computed separately, it should not change the analysis herein described.

However, if we really need to be conservative, loads of instructions may throw runtime exceptions (IndexOutOfBoundsException, NullPointerException, etc.) or even errors (OutOfMemoryError, etc.) and so there will be edges in the control flow graph from most program points to the exit point of the methods, making the analysis very imprecise.

There are several ways to improve the precision while safely handling exceptions. First, we can prove the absence of exception for some of those (e.g. see [9] to remove most NullPointerExceptions and [1] to remove the IndexOutOfBoundsException you need to remove). Then it is cheap to analyse, for each method, the context in which the method is called, i.e. the exceptions that may be caught if they are thrown by the method. If there are no handler for some exception in the context of a particular method, then there is no use to take in account this exception in the control flow graph of this method. Indeed, if such an exception were thrown it would mean the termination of the program execution so not taking in account the exception may only add potential behaviours, which is safe.

5.2 Inheritance

In the presence of a class hierarchy, the initialization of a class starts by the initialization of its superclass if it has not been done yet. There is therefore an implicit edge in the control flow graph from each \(<\text{cinit}>\) method to the \(<\text{cinit}>\) method of its superclass (except for \text{Object.<cinit>}). Although it does not involve any challenging problem, the semantics and the formalization need to be modified to introduce a new label at the beginning of each \(<\text{cinit}>\) method such that, if \(f\) is the label we introduce at the beginning of \(C.<\text{cinit}>\), then \(\text{flow}_{\text{initial}}(f) = \text{super}(C)\).

Note that it is not required to initialize the interfaces a class implements, nor the super interfaces an interface extends (cf. Sect. 2.17.4 of [12]).

### Resource usage equations

<table>
<thead>
<tr>
<th>Resource usage equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Cost((s, r_{x_0}, s_{x_1}, s_{x_2}, s_{x_3}) ≤ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) )]</td>
</tr>
<tr>
<td>[+ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) ]</td>
</tr>
<tr>
<td>[+ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) ]</td>
</tr>
<tr>
<td>[+ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) ]</td>
</tr>
<tr>
<td>[+ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) ]</td>
</tr>
<tr>
<td>[+ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) ]</td>
</tr>
<tr>
<td>[+ \min(\text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3), A_{in}(s, h) + \text{lookup}(mt, cost, $, sr_0, sr_1, sr_2, sr_3)) ]</td>
</tr>
<tr>
<td>[\leq 12 \times (s_{x_1} - 1) + \text{Cost}(s_{x_2}, s_{x_1} - 1, s_{x_2}, s_{x_3})]</td>
</tr>
</tbody>
</table>

**Simplified resource usage equations and closed form solution**

\[
\begin{align*}
\text{Cost}(s, r_{x_0}, 0, s_{x_1}, s_{x_2}, s_{x_3}) & \leq \min(\text{lookup}(mt, cost, $, eq), A_{in}(s, h)) \\
& + \min(\text{lookup}(mt, cost, $, eq), A_{in}(s, h)) \\
& \leq 0 \\
\text{Cost}(s, r_{x_0}, 0, s_{x_1}, s_{x_2}, s_{x_3}) & \leq 6 \times s_{x_0} - 6 \times s_{x_1} \\
\text{Cost}(s_{x_0}, s_{x_1}, s_{x_2}, s_{x_3}) & \leq 0 \\
\text{Cost}(s, r_{x_0}, 0, s_{x_1}, s_{x_2}, s_{x_3}) & \leq 0 \\
\text{Cost}(s, r_{x_0}, 0, s_{x_1}, s_{x_2}, s_{x_3}) & \leq 0 \\
\end{align*}
\]

Fig. 7. Resource equations example.

block method matches the signature of the block method itself and thus it is recursive. The first four parameter positions are of input type. The upper-bound expression returned by \text{genConstraint} is \(Cost((\text{cents}, min(s_{x_1} - 1))\) since the input parameter at position one is at most six times the size of the second input formal parameter, as calculated by the size analysis in Fig. 5. Note also that there is a resource annotation \@Cost("(\text{cents},"",\text{size(r)}")\) attached to the block method describing the behavior of \(A_{cond}\) and yielding the expression \(\text{Cost}(\text{user}) = 12 \times (s_{x_1} - 1)\). On the other hand, the absence of any callee code to analyze – the original method is native – results in \(\text{Cost}(\text{dy}) = \infty\). Then, the upper bound obtained by the analysis for the statement is \(\min(\text{Cost}(\text{dy}), \text{Cost}(\text{user})) = \text{Cost}(\text{user})\).

At this point, the analysis has built a resource usage function (denoted by \text{Costbody}) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form \(\text{Cost}(\text{block}) \leq \text{Cost}(\text{dy})\) where \(\text{Cost}(\text{block})\) is again a symbolic resource usage function built by replacing each input formal parameter position with its size relations in that block method. These resource usage equations are simplified by calling \text{reduce} and, finally, they are solved calling \text{reduce} again, both already defined in Sec. 4.1. This process yields an (in gen-
5.3 Field Initializers and Initialization order

Although the official JVM specification states (Sect. 2.17.4) that the initialization of the superclass should be done before the initialization of the current class and that the field initializers are part of the initialization process, in Sun’s JVM the field initializers are used to set the corresponding fields before starting the initialization of the superclass. This changes the semantics but removes potential defects as this way it is impossible for some code to read the field before it has been set.

If the analysis is targeted to a such JVM implementation, depending on the application of the analysis, the fields which have a field initializer can be safely ignored when displaying the warnings found, or be added to HJ either when the corresponding class is added to $\text{Must}$ or at the very beginning ($m_{\text{first}}$). In order for the analysis to be compatible with the official specification, the analysis needs to simulate the initialization of the fields that have a field initializer at the beginning of the class initializer, just after the implicit call to the class initializer of its superclass.

5.4 Reflection, User-Defined Class-Loader, Class-Path Modification, etc.

There are several contexts in which this analysis can be used: it can be used to find bugs or to prove the correctness of some code, either off-line or in a PCC [15] architecture.

In case it is used to find bugs, the user mainly need to be aware that those features are not supported. If it is used to prove at compile time the correctness of some code, the analysis needs to handle those features. In case of reflection, user-defined class-loaders or modification of the class-path, it is difficult to be sure that all the code that may be executed has been analyzed. The solution would restrict the features of the language in order to be able to infer what code may be executed, which is an over-approximation of the code that will be executed, and to analyze this code. For example, Livshits et al. proposed in [13] an analysis to correctly handle reflection. In a PCC architecture, the code is annotated at compile time and checked at run time. The issue is no more to find the code that will be executed, because the checking is done at run time when it is a lot easier to know what code will be executed, but to consider enough code when annotating. This can be done the same way as with off-line proofs and by asking the programmer when it is not possible to infer a precise enough solution. In this case, if the user gives incorrect data the checker will notice it while checking the proof at run time.

6 Two Possible Applications for the Analysis

6.1 Checking That Fields are Written Before Being Read

The analysis presented in this paper computes a set of fields that have been written for each program point. To check that all fields are written before being read, i.e. that when a field $\text{C.f}$ is read at program point $p$, we need to check that $\text{C.f}$ is in the set of written field at this particular program point.
6.2 Nullness Analysis of Static Fields

While in our previous work [10] we choose to assume no information about static fields, several tools have targeted the analysis of static fields as part of their analysis but missed the issue of the initialization herein discussed.

To safely handle static fields while improving the precision, we can abstract every field by the abstraction of the values that may be written to it and, if the static field may be read before being initialized, then we add the abstraction of the null constant to the abstraction of the field. It is a straightforward extension of the analysis presented in [10].

7 Related work

Kozen and Stillerman studied in [11] eager class initialization for Java bytecode and proposed a static analysis based on fine grained circular dependencies to find an initialization order for classes. If their analysis finds an initialization order, then our analysis will be able to prove all fields are initialized before being read. If their analysis finds a circular dependency, it fails to find an initialization order and issues an error while our analysis considers the initialization order implied by the main program and may prove that all fields are written before being read.

Instance field initialization have been studied for different purposes. Some works are focused on null-ability properties such as Fähndrich and Lehbo in [5], our work in [10] or Fähndrich and Xia in [6]. Other work have been focused on different properties such as Uhkel and Lam [16] who studied stationary fields. Instance field initialization offers different challenges from the one of static fields: the initialization method is explicitly called soon after the object allocation.

Several formalizations of the Java bytecode have been proposed that, among other features, handled class initialization such as the work of Debhabh et al. in [4] or Belhadi and Debhab in [14]. Their work is focused on the dynamic semantics of the Java bytecode while our work is focused on its analysis.

Böerger and Schulte [2] propose another dynamic semantics of Java. They consider a subset of Java including initialization, exceptions and threads. They have exhibited [3] some weaknesses in the initialization process as far as the threads are used. They pointed out that deadlocks could occur in such a situation.

Harrold and Sofía [7] propose an analysis to compute inter-procedural definition-use chains. They have not targeted the Java bytecode language and therefore neither the class initialization problem we have faced but then, our analysis can be seen as a lightweight inter-procedural definition-use analysis where all definitions except the default one are merged.

Hirzel et al. [8] propose a pointer analysis that target dynamic class loading and lazy class initialization. There approach is to analyse the program at run time, when the actual classes have been loaded and to update the data when a new class is loaded and initialized. Although it is not practical to statically certify programs, a similar approach could certainly be adapted to implement a checker in PCC architecture such as the one evoked in Sect. 5.4.

Size parameter relationship equations (normalized)

\[
\begin{align*}
\text{size}(ref, ne, 0) & \leq \text{size}(ref, \text{sendSms}, 1) \leq s_{r1} \\
\text{size}(ref, ne, 1) & \leq \text{val}(ref, stf, 0) \leq s_{r1} \\
\text{size}(ref, stf, 0) & \leq \text{size}(ref, stf, 1) \leq s_{r1} \\
\text{size}(ref, stf, 1) & \leq \text{size}(ref, stf, 2) \leq s_{r1} \\
\text{size}(ref, stf, 2) & \leq \text{size}(ref, stf, 3) \leq s_{r1} \\
\text{size}(ref, stf, 3) & \leq \text{size}(ref, stf, 0) \leq s_{r1} \\
\end{align*}
\]

Output parameter size functions for builtins (provided through annotations)

\[
\begin{align*}
A_{\text{ref}}(\text{size}(ref, stf, 1)) & \leq \text{size}(ref, stf, 0) - 1 \\
A_{\text{ref}}(\text{size}(ref, stf, 0)) & < \text{size}(ref, stf, 0) \\
A_{\text{ref}}(\text{size}(ref, stf, 0)) & < \text{size}(ref, stf, 0) \\
\end{align*}
\]

Simplified size equations and closed form solution

\[
\begin{align*}
\text{Size}_{\text{ref}}^{\text{Kozen-Stillerman}}(ref, stf, 0, stf, 1, stf, 2) & \leq \begin{cases} 
0 & \text{if } s_{r1} = 0 \\
7 + s_{r1} - 6 + \text{Size}_{\text{ref}}^{\text{Kozen-Stillerman}}(ref, stf, 0, stf, 1, stf, 2) & \text{otherwise} 
\end{cases} \\
\text{Size}_{\text{ref}}^{\text{Kozen-Stillerman}}(ref, stf, 0, stf, 1, stf, 2, stf, 3) & \leq s_{r1} - 6 + \text{Size}_{\text{ref}}^{\text{Kozen-Stillerman}}(ref, stf, 0, stf, 1, stf, 2, stf, 3) \\
\end{align*}
\]

Fig. 5. Size equations example
which results in equation \( A_{\text{asg}}^1(\text{ref}, \text{size}(\text{ref}, \text{asg}, 0)) \leq \text{size}(\text{ref}, \text{asg}, 0) \).

If the output parameter position corresponds to a recursive invoke statement, the size relationships between the output and input parameters are built as a symbolic size function. Since the input parameter size relations have already been computed, we can establish each output parameter position size as a function described in terms of the input parameter sizes.

At this point, the algorithm has defined size relations for all parameter positions within a block method. However, these relations are either constants or given in terms of the immediate predecessor in the dependency graph. The algorithm rewrites the equation system such that we obtain an equivalent system in which only formal parameter positions are involved. This process, called normalization, is shown in Fig. 2.

After normalization, the analysis repeats the same process for all block methods in the same strongly-connected component (SCC). Once every component has been processed, the analysis further simplifies the equations in order to resolve mutually recursive calls among block methods within the same SCC in the simplest procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver \((\text{genSizeSolver}, \text{called from genSizes})\) in order to obtain approximated upper-bound closed forms. The interesting subject of how the equations are solved is beyond the scope of this paper (see, e.g., [41]). Our implementation does provide a simple built-in solver (an evolution of the solver of the Caslog system [16]) which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients, divide and conquer recurrence equations, etc. However, it also includes an interface to the Parma Polyhedra Library \([7]\) (and previously to other tools such as Mathematica, Matlab, etc.). Work is also under way to interface with the quite interesting solver of [1].

Example 3 We now illustrate the definitions and algorithms with an example of how the size relations are inferred for the two \texttt{CellPhone.sendSms} block methods (Fig. 1), using the \texttt{ref} measure for reference variables. We will refer to the \(k\)-th occurrence of a statement \(\text{stmt}\) in a block method as \(\text{stmt}_k\), and denote \texttt{CellPhone.sendSms}, \texttt{Encoder.format}, and \texttt{Stream.send} by \texttt{sendSms}, \texttt{format}, and \texttt{send} respectively. Finally, as mentioned before, we refer to the size of the input formal parameter position \(i\), corresponding to variable \(r_i\), as \(s_i\).

The main steps in the process are listed in Fig. 5. The first block of rows contains the most relevant size parameter relationship equations for the recursive block method, while the second block of rows corresponds to the base case. These size parameter relationship equations are constructed by the analysis by first following the algorithm in Fig. 4, and then normalizing them (expressing them in terms of the input formal parameter sizes \(s_{i}\) ). Also, in the first block of rows we observe that the algorithm has returned \(6 \times \text{size}(\text{ref}, \text{format}, 1)\) as upper bound for the size of the formatted string, \(\text{max}(\text{looks}(\text{ref}, \text{size}, 1))\). The result is

8 Conclusion and Future Work

We have shown that class initialization is a complex mechanism and that, although in most cases it works as expected, in some more complicated examples it can be complex to understand in which order the code will be executed. More specifically, some fields may be read before being initialized, despite being initialized in their corresponding class initialization methods. A sound analysis may need to address this problem to infer precise and correct information about the content of static fields. We have proposed an analysis to identify the static fields that may be read before being initialized and shown how this analysis can be used to infer more precise information about static fields in a sound null-pointer analysis.

We expect the analysis to be very precise if the control flow graph is accurate enough, but we would need to implement this analysis to evaluate the precision needed for the control flow graph.

References

genSizeEqs(SCC, mt, CFG, dg) {
Eqs ← ∅ | SCC |
for (sig : SCC)
Eqs[sig] ← genSigSize(sig, mt, SCC, CFG, dg)
return Eqs
}

genSigSize(sig, mt, SCC, CFG, dg) {
Eqs ← ∅
for (bm : BMs)
Eqs←genBlockSize(bm, mt, SCC, dg)
return Eqs
}

genBlockSize(bm, mt, SCC, dg) {
Eqs ← ∅
for (stmt : bm.body)
Eqs←genInSize(stmt, mt, SCC, dg)
Eqs←genOutSize(stmt, mt, SCC)
Eqs←recEqsSolver(simplifyEqs(Eqs))
return Eqs
}

genInSize(elem, Pos, mt, dg) {
Eqs ← ∅
for (pos : Pos)
Eqs←getEqs(elem, sig, pos)
return Eqs
}

genOutSize(stmt, mt, SCC) {
Eqs ← ∅
for (stmt : mt.inputParameterPositions)
Eqs←getEqs(stmt, mt, SCC)
Eqs←recEqsSolver(simplifyEqs(Eqs))
return Eqs
}

Example 2
We have already shown in the CellPhone example how a class can be annotated. The BuiltIn class includes the assignment method asg, annotated as follows:

```
public class BuiltIn {
    @Size("size(ret) ≤ size(o)")
    public static native Object asg(Object o);
}
```
User-Definable Resource Usage
Bounds Analysis for Java Bytecode

Jorge Navas$^{1,4}$
1School of Computing
National University of Singapore
Republic of Singapore

Mario Méndez-Lojo$^{2,4}$
2Department of Computer Science
University of New Mexico, Albuquerque, NM (USA)

Manuel V. Hermenegildo$^{3,5}$
3IMDEA Software, Madrid (Spain)
Departments of Computer Science
Universidad de Cantabria, Santander, Spain and
3Technical University of Madrid, Madrid (Spain)

Abstract
Automatic cost analysis of programs has been traditionally concentrated on a reduced number of resources such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, the number of objects and references used, the number of SMSs sent or received, number of accesses to a database, memory space, energy consumption, etc. We present a fully automated analyser for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis is based on an intermediate representation of the bytecode which captures all resource-related information. The prototype implementation of the approach covering a significant set of interesting resources resulted from a paper published in Theoretical Computer Science.

1 Introduction
The usefulness of analyses which can infer information about the costs of computations is widely recognized. Such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [27,34,36,29,21,40] for functional languages, [38,8,19,42] for imperative languages, and [17,18,32] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [29]), resource-oriented specialization (e.g., [13,33]), or, more recently, certification of the resources used by mobile code such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, the number of objects and references used, the number of SMSs sent or received, number of accesses to a database, memory space, energy consumption, etc. We present a fully automated analyser for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis is based on an intermediate representation of the bytecode which captures all resource-related information. The prototype implementation of the approach covering a significant set of interesting resources resulted from a paper published in Theoretical Computer Science.

1 Introduction
The usefulness of analyses which can infer information about the costs of computations is widely recognized. Such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [27,34,36,29,21,40] for functional languages, [38,8,19,42] for imperative languages, and [17,18,32] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [29]), resource-oriented specialization (e.g., [13,33]), or, more recently, certification of the resources used by mobile code such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, the number of objects and references used, the number of SMSs sent or received, number of accesses to a database, memory space, energy consumption, etc. We present a fully automated analyser for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis is based on an intermediate representation of the bytecode which captures all resource-related information. The prototype implementation of the approach covering a significant set of interesting resources resulted from a paper published in Theoretical Computer Science.

1 Introduction
The usefulness of analyses which can infer information about the costs of computations is widely recognized. Such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [27,34,36,29,21,40] for functional languages, [38,8,19,42] for imperative languages, and [17,18,32] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [29]), resource-oriented specialization (e.g., [13,33]), or, more recently, certification of the resources used by mobile code such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, the number of objects and references used, the number of SMSs sent or received, number of accesses to a database, memory space, energy consumption, etc. We present a fully automated analyser for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis is based on an intermediate representation of the bytecode which captures all resource-related information. The prototype implementation of the approach covering a significant set of interesting resources resulted from a paper published in Theoretical Computer Science.
resourceAnalysis(CFG, res) { 
CFG ← classAnalysis(CFG)
Aliases ← aliasAnalysis(CFG) 
mt ← initialize(CFG)
dg ← dataDependencyAnalysis(CFG, Aliases, mt)
for (SCC: CFGs) 
// in reverse topological order 
mt ← genSizeEq(CFG, mt, dG)
mt ← ResourceUsageEq(CFG, res, mt) 
return mt 
} 

Fig. 2. Generic resource analysis algorithm and normalization.

4 The resource usage analysis framework

We now describe our framework for inferring upper bounds on the usage that the Java bytecode program makes of a set of resources defined by the application programmer, as described before. The algorithm in Fig. 2 takes as input a Control Flow Graph in the format described in the previous section, including the user annotations that assign elementary costs to certain graph elements for a particular resource. The user also indicates the set of resources to be tracked by the analysis.

Without loss of generality, we assume for conciseness in our presentation a single resource.

A preliminary step in our approach is a class hierarchy analysis [15,30], aimed at simplifying the CFG and therefore improving overall precision. More importantly, we also require the existence of an alias analysis [35,26,11], whose results are used by a third phase (described below) in which data dependences between variables in the CFG are inferred. The next step is the decomposition of the CFG into its strongly-connected components. After these steps, two different analyses are run separately on each strongly connected component: a) the size analysis, which estimates parameter size relationships for each statement and output formal parameters as a function of the input formal parameter sizes (Sec. 4.1); and b) the actual resource analysis, which computes the resource usage of each block method in terms also of the input data sizes (Sec. 4.2). Each phase is dependent on the previous one.

The data dependency analysis is a dataflow analysis that yields position dependency graphs for the block methods within a strongly connected component. Each graph \( G = (V, E) \) represents data dependences between positions corresponding to statements in the same block method, including its formal parameters. Vertices in \( V \) denote positions, and edges \((s_1, s_2) \in E \) denote that \( s_2 \) is dependent on \( s_1 \). We will assume a \texttt{predc} function that takes as input a position dependency graph, a statement, and a parameter position and returns its nearest predecessor in the graph. Fig. 3 shows the position dependency graph of the \texttt{TrimEncoder.format} block method.

4.1 Size analysis

We now show our algorithm for estimating parameter size relations based on the data dependency analysis, inspired by the original ideas of [17,16]. Our goal is
The Control Flow Graph is composed of block methods. A block method is similar to a Java method, with some particularities: a) if the program flow reaches it, every statement in it will be executed, i.e. it contains no branching; b) its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types; c) it always includes as formal parameters the returned value ret and, if it is static, the instance self-reference this; d) for every formal parameter (input formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the block method that contains its final version in the SSA transformation (output formal parameter); e) every statement in a block method is an invocation, including builtins (assignment =, field dereference .get(), etc.), which are understood as block methods of the class Builtin.

As mentioned before, there is no branching within a block method. Instead, each conditional if cond stmt1 else stmt2 in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the stmt1 branch, and the second one to stmt2. To respect the semantics of the language, we decorate the first block method with the result of the decomposition cond, while we attach cond to its sibling. A similar approach is used in virtual invocations, for which we introduce as many block methods in the graph as possible receivers of the call were in the original program. A set of block methods with the same signature sig can be retrieved by the function getBlocks(CFG, sig).

User specifications are written using the annotation system introduced in Java 1.5 which, unlike JML specifications, has the very useful characteristic of being self-referential: it allows one to make of the resource. For example, it is not intended to be realistic, but rather a small piece of code that illustrates a number of aspects of the approach. The source code is one containing a single SMS, encodes them (Encoder.format(String data)), and can be retrieved by the function getBlocks(CFG, sig).

Example 1 We now focus our attention on the two block methods in Fig. 1, which are the result of (de)compiling the CellPhone.sendSMS method. The input formal parameters r0, r1, r2, r3 correspond to this, smsPk, enc, and stmt, respectively. In the case of r2, the contents of its fields next and sms are altered by invoking the stf and accessed by invoking the gtf (abbreviation for setfield and getfield, respectively) built in block methods. The output formal parameter r4 contains the final state of r2 after those modifications. The value returned by the block methods is contained in r4. Space reasons prevent us from showing any type information in the CFG in Fig. 1. In the case of Encoder.format, for example, we say that there are two blocks with the same signature because they are both defined in class Encoder, have the same name (format) and the same list of types of formal parameters (Encoder(String, String)).
any implementation of Encoder.format is free, as indicated by the @Cost-
(("cents","0")}) annotation (the actual system allows defining overall cost de-
tails but we express them here explicitly). The analysis then processes these local
resource usage expressions and uses them to infer a safe upper bound on the total
(global) usage of the defined resources made by the program.

As illustrated by the example, these Java annotations allow defining the re-
sources to be tracked (which is done by simply mentioning them in the annota-
tions) and to provide cost functions for the built-in and external (library) blocks
that are relevant to the particular resource (i.e., which affect the usage of such re-
source). They also allow defining data size relations among arguments and defining
and declaring size measures. The resource usage expressions are defined using the
following language (which we will call \(\mathcal{L}\)):
\[
\begin{align*}
\langle \text{expr} \rangle & := \langle \text{expr} \rangle/\langle \text{expr} \rangle | \sum \langle \text{expr} \rangle | \prod \langle \text{expr} \rangle \\
& | \langle \text{expr} \rangle^{+} | \langle \text{expr} \rangle^{-} | \langle \text{expr} \rangle | \langle \text{expr} \rangle! \\
& | \langle \text{expr} \rangle dik | \langle \text{expr} \rangle\ | \langle \text{expr} \rangle\ | \langle \text{expr} \rangle\ |
\end{align*}
\]
\[
\begin{align*}
\langle \text{bin} \rangle & := + | - | \times | / | % \\
\langle \text{measure} \rangle & := \text{int} | \text{ref} \ldots
\end{align*}
\]

We now summarize the fundamental steps of the analysis:

**Step 1: Constructing the Control Flow Graph.**

In the first step, the analysis translates the Java bytecode into an intermediate
representation building a Control Flow Graph (CFG). Edges in the CFG connect
block methods and describe the possible flows originated from conditional jumps,
exception handling, virtual invocations, etc. A (simplified) version of the CFG
corresponding to our code example is also shown in Fig. 1.

The original sendSms method has been compiled into two block methods that
share the same signature: class where declared, name (CellPhone.sendSms), and
number and type of the formal parameters. The bottom-most box represents the
base case, in which we return null, here represented as an assignment of null to
the return variable \(r_5\); the sibling corresponds to the recursive case. The virtual
invocation of format has been transformed into a static call to a block method
named Encoder.format. There are two block methods which are compatible
in signature with that invocation, and which serve as proxies for the intermedi-
ate representations of the interface implementations in TrimEncoder.format and
UnicodeEncoder.format. Note that the resource-related annotations have been
carried through the CFG and are thus available to the analysis.

**Step 2: Inference of Data Dependencies and Size Relationships.**

The algorithm infers in this phase size relationships between the input and the
output formal parameters of every block method. We assume that the size of (the
contents of) a linked structure pointed to by a variable is the maximum number
of pointers we need to traverse, starting at the variable, until null is found. The
following equations are inferred by the analysis for the two CellPhone.sendSms
block methods (with \(r_i\) denote the size of input formal parameter position \(i\),
 corresponding to variable \(r_i\):
\[
\begin{align*}
\text{Sizer5} \text{sendSms}(sr_0, 0, sr_2, sr_3) & \leq 0 \\
\text{Sizer5} \text{sendSms}(sr_0, sr_1, sr_2, sr_3) & \leq 7 \times sr_1 - 6 + \text{Sizer5} \\
\text{sendSms}(sr_0, sr_1, sr_2, sr_3) & \leq 7 \times sr_1 - 6 + \text{Sizer5} \\
\text{sendSms}(sr_0, sr_1, sr_2, sr_3) & \leq 7 \times sr_1 - 6 + \text{Sizer5}
\end{align*}
\]

Step 3: Resource Usage Analysis.

In this phase, the analysis uses the CFG, the data dependencies, and the size
relationships inferred in previous steps to infer a resource usage equation for each
block method in the CFG (possibly simplifying such equations) and obtain closed
form solutions (in general, approximated – upper bounds). Therefore, the objective
of the resource analysis is to statically derive safe upper bounds on the amount
of resources that each of the block methods in the CFG consumes or provides.
The result given by our analysis for the monetary cost of sending the messages
(CellPhone.sendSms) is
\[
\begin{align*}
\text{CostsendSms}(sr_0, sr_1, sr_2, sr_3) & \leq 0 \\
\text{CostsendSms}(sr_0, sr_1, sr_2, sr_3) & \leq 12 \times sr_1 - 12 + \text{CostsendSms}(sr_0, sr_1, sr_2, sr_3)
\end{align*}
\]
i.e., the cost is proportional to the size of the message list (smsPk in the source, \(r_1\)
in the CFG). Again, this equation system is solved by a recurrence solver, resulting
in the closed formula
\[
\text{CostsendSms}(sr_0, sr_1, sr_2, sr_3) \leq 6 \times s_5^2 - 6 \times sr_1.
\]

3 Intermediate program representation

Analysis of a Java bytecode program normally requires its translation into an inter-
mediate representation that is easier to manipulate. In particular, our decompilation
(assisted by the Soot [39] tool) involves elimination of stack variables, conversion
to three-address statements, static single assignment (SSA) transformation, and
generation of a Control Flow Graph (CFG) that is ultimately the subject of analysis.
Note that in this representation loops are converted into recursive blocks. The de-
compilation process is an evolution of the work presented in [31], which has been
successfully used as the basis for other (non resource-related) analyses [30]. Our
ultimate objective is to support the full Java language but the current transforma-