Preproceedings of

BYTECODE 2009

4^{th} International Workshop on Bytecode Semantics, Verification, Analysis and Transformation

http://www.clip.dia.fi.upm.es/Conferences/BYTECODE09

Satellite event of ETAPS 2009

York, UK ,29^{th} March 2009

Preface

This volume contains the proceedings of the Bytecode 2009 workshop, the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation, held in York, UK, on the 29th of March 2009 as part of ETAPS 2009.

Bytecode, such as produced by e.g. Java and .NET compilers, has become an important topic of interest, both for industry and academia. The industrial interest stems from the fact that bytecode is typically used for the Internet and mobile devices (smartcards, phones, etc.), where security is a major issue. Moreover, bytecode is device-independent and allows dynamic loading of classes, which provides an extra challenge for the application of formal methods. In addition, the unstructuredness of the code and the pervasive presence of the operand stack also provide extra challenges for the analysis of bytecode. This workshop focuses on the latest developments in the semantics, verification, analysis, and transformation of bytecode; encompassing both new theoretical results and tool demonstrations. There were 16 submissions. Each submission was reviewed by at least 3 programme committee members. The committee decided to accept 11 papers. The programme also includes 1 invited talk by Thomas Jensen.

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

Conference Organization

Programme Chairs

Elvira Albert
Samir Genaim

Programme Committee

Wolfgang Ahrendt
June Andronick
David Aspinall
Cristina Cifuentes
Sara Kalvala
Gerwin Klein
Francesco Logozzo
David Pichardie
Tamara Rezk
Fausto Spoto
Eran Yahav

External Reviewers

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

Contents

Session 1

Session 2

Session 3

Session 4

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

Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

Frédéric Besson, Thomas Jensen, Tiphaine Turpin. Small witnesses for abstract interpretation based proofs. In *Proceedings of the 16th European Symp. on Programming (ESOP 2007)*, Springer LNCS vol. 4421, 2007.

Frédéric Besson, Thomas Jensen, Tiphaine Turpin. Computing Stack Maps with Interfaces, In *Proc. of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, Springer LNCS vol. 5142.

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

Pervasive Load-Time Transformation for Transparently Distributed Java

Phil McGachey[a] Antony L. Hosking[a] J. Eliot B. Moss[b]

[a] *Department of Computer Sciences, Purdue University, West Lafayette, IN, USA*

[b] *Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, USA*

Abstract

The transformation of large, off-the-shelf Java applications to support complex new functionality essentially requires generation of 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

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 be 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

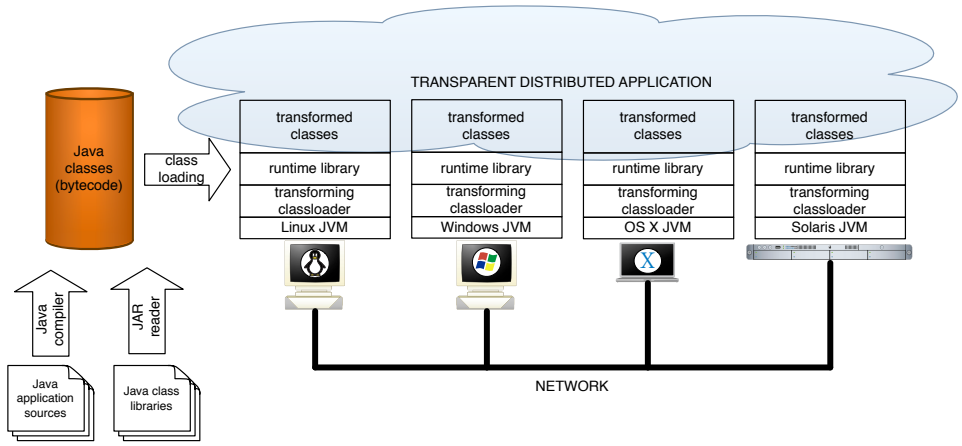


Fig. 1. The RuggedJ System Architecture

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 standard RuggedJ node, the head node acts 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

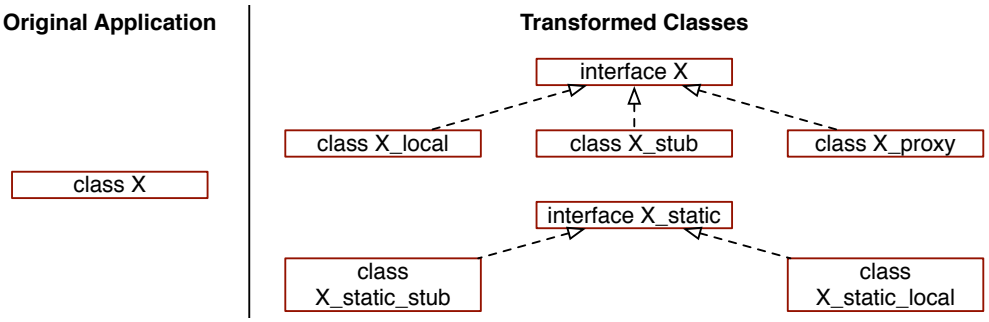


Fig. 2. Classes generated by the RuggedJ class loader

of the library are called by rewritten code, while others coordinate to provide networking services and to generate an accurate picture of the network as a whole.

Library Functionality: The run-time library provides functionality to rewritten bytecode. Many of the operations required to support distribution require complex code sequences. Specifying such operations within a rewritten method would very quickly lead to unreadable bytecode, which is difficult to debug. Instead, we delegate to the run-time library all operations that require more than a very simple bytecode sequence.

Run-time Co-ordination: There are several key tasks performed by the run-time to co-ordinate execution between nodes. These include monitoring the network and host configurations and status, tracking the location of objects for remote method calls and message passing between nodes.

3 The RuggedJ Object Model

We accomplish distribution in RuggedJ by abstraction of object locations. We achieve this through use of proxies for objects. Proxies allow the implementation of an object to vary depending on whether it is local or remote, while presenting a single interface to external code. For every class in the original application, RuggedJ generates a series of classes and interfaces, as shown in Figure 2.

We split classes into two parts: the fields and methods that make up per-object state (the instance parts), and those that are specific to the class (the static parts). This is necessary in order to ensure that static data exists exactly once in the RuggedJ network; Section 3.3 discusses this issue further.

When the RuggedJ class loader has rewritten a class, it presents only the transformed version for loading into the Java VM. The VM never sees the original class, which removes the possibility of conflicts between modified and unmodified classes. The only exception to this is in the case of unmodifiable classes, which we describe in Section 3.4.

3.1 Instance Classes

Focusing on the instance parts of Figure 2, we see the three classes and one interface that RuggedJ generates from the instance parts of each application class **X**:

Interface X: The interface contains an abstract version of each instance method present in the original application class, as well as get and set methods for each field (see Section 4.2). The name of the interface is significant. By using the name of the original

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. 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

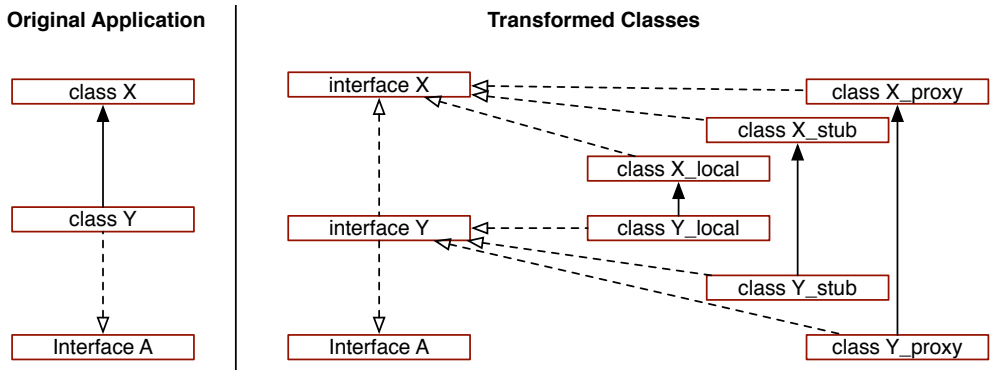


Fig. 3. Inheritance between transformed classes

objects exclusively by interface, this allows one to use any object that implements `Y` when the original code required an instance of `X`. 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_local`, `Y_stub` and `Y_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_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_static_local` and `X_static_stub` classes implement `X_static`, allowing rewritten code to use them interchangeably.

Class X_static_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_static_local` rather than static members. This allows class `X_static_local` to fulfill the requirements of interface `X_static`, and decouples the implementation of static members from the implementation of the VM.

Class X_static_stub: The static stub acts similarly to the instance stub class. It contains implementations of each method in interface `X_static` that perform remote invocations on the appropriate `X_static_local` object.

Transforming static methods of original class `X` into instance members of class `X_static_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

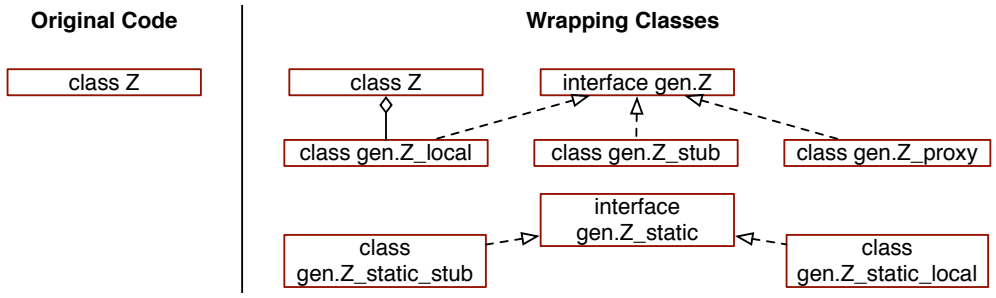


Fig. 4. Wrapping unmodifiable classes

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_static_local`. Once some node has allocated the singleton, all other nodes will create `X_static_stub` objects as required. By managing static singletons through the run-time, we eliminate the need for a `X_static_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 `gen.Z_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 `gen.Z_stub` and

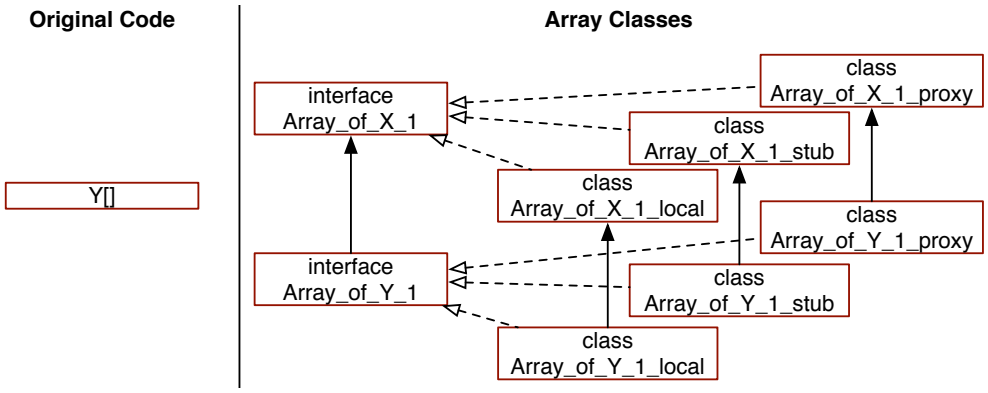


Fig. 5. Generated array classes

`gen.Z_proxy` identically to the stub and proxy classes described in Section 3.1.

Class `gen.Z_static_local` acts as a static singleton for the wrapped class, with one important difference. Since an unmodifiable class may directly access static members, we cannot rewrite such static data to form instance methods of `gen.Z_static_local`. Thus, the methods of the static local class instead simply delegate to the original class. To ensure uniqueness in static data, a given unmodifiable class can thus access static data only on a single node. While this limits the potential for distribution, in practice most such classes tend to be allocated only on the head node, with transformable application classes running on all other nodes.

Rewritten code interacts with unmodifiable code using the interfaces and wrapping classes in the same way as regular classes. This allows the same transparency with regard to object location for wrapped classes that exists for transformed classes. However, since the unmodifiable code itself is unaware of transformed classes, we must unwrap objects when passing them as arguments to unmodifiable code, and re-wrap returned objects. The unwrapping process is simple: the methods with `Z_local` unwrap any transformed class arguments (recall that, by definition, all classes passed to unmodifiable code are themselves unmodifiable and so can be unwrapped). Wrapping return values is slightly more difficult, since we must ensure that a given object has only one wrapper—an object returned multiple times from unmodifiable code must always be wrapped by the same object. We control this in the RuggedJ run-time library, which tracks generated wrappers and creates a new wrapper only if the object has not been wrapped before.

3.5 Arrays

When distributing an application, we must transform not only objects but also arrays. To this end, we generate classes for array types, 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. Class `Array_of_Y_1_local` is a wrapper for a one-dimensional array of `Y` objects (the contents of which are themselves instances of local, stub, or proxy classes that implement interface `Y`). Array classes do not need static singletons, since arrays maintain no static state.

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 `Array_of_Y_2` and so forth), with the local class containing a wrapped array. The wrapped array is always one-dimensional, so `Array_of_Y_2_local` contains an array of `Array_of_Y_1` 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, `java.lang.System` contains several methods for which we define special semantics: we must redirect all references to `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 knot 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

simultaneously creating singletons. When the code has found the singleton, the modified bytecode can then invoke the necessary method on it. In the case of field accesses, this consists of a call to the appropriate get or set method.

This is clearly a costly operation, particularly in the case where the static singleton is a stub, and a method invocation requires access to a remote object. As such, we minimize access to static singletons as far as possible. We observe that the static singleton exists only to ensure that there is only one copy of static data. Therefore, we need to call the singleton only when we may access that state: calls to static methods that do not read or write the singleton's fields do not go through the singleton. Rather, they call a local version of the static method. Indeed, for classes with no static state, it is not necessary to create a static singleton at all.

Initialization of static data is performed by the run-time system when a static singleton is created. Any `static{}` code block is transformed into an instance method, allowing it to be called at the appropriate time during singleton creation. Static final fields (constants) are treated in the same way as all other static fields; constants are initialized by the static initializer, and can have different values on different nodes (consider a constant initialized to a host's IP address). Forcing static final fields to go through a static singleton is conservative, and can be optimized in many cases.

4.4 Allocation

The object allocation process involves interaction between the rewritten bytecode in a method and the partitioning strategy defined by the application author. It is the primary means by which one distributes an application. By strategically allocating objects on remote nodes and remotely invoking methods, one can perform large computations across a collection of nodes.

We define an *allocation site* as an instance of a `new` bytecode. When rewriting an allocation site, the RuggedJ rewriting class loader first queries the partitioning plug-in with static site information to request a load-time allocation strategy. The allocation site information includes the class and method in which the allocation site occurs and the type it allocates. Based on this, the partitioning can return one of three options:

Allocate Locally: If the policy knows that the code uses the particular type of object principally on the local node, we can streamline the allocation process to create the local version of the class. This is a fairly common case: some classes rely on local resources, many objects are temporary and of purely local interest, and domain-specific knowledge may determine that an object will rarely be used by another node. The partitioning plug-in may also determine whether to allocate a proxy to allow for later migration, or simply to allocate the local version directly, allowing it to be remotely referenced but not migrated.

Allocate Remotely: On the other hand, a policy may sometimes know that we should always allocate an object on a different node. This may be the case if the partitioning strategy dictates to spread objects of a certain across the network for load balancing purposes, or that a particular class would benefit from a resource that is not available on the local host. This option allocates both a proxy and a stub object, and determines at run-time on which node to allocate the object.

Allocate Dynamically: Finally, there are cases where we will not know the best allocation node for an object until run time. This may be the case if we should evenly distribute

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 scan 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 (`aaload`, `aastore`, `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 `invokeinterface` method calls, we need to know both the type and dimensionality of the array upon which to invoke the method. We find this information using a 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 `aload_0` 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

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 `CallStatic<type>Method` 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

the object model implemented by a transformed application running under the RuggedJ transparent distribution system. We have outlined the classes and interfaces required for a flexible, dynamic distributed system and described how such an object model maintains the semantics of the original application. We then discussed the process of transforming an application to implement this object model, including the classes generated, modification to method bodies, and the dynamic distribution of an application though object allocation.

We believe that the techniques described in this paper offer insight into some of the issues involved in large-scale transformation of Java applications, and may serve to guide future implementations not only of distributed Java but of any system that uses indirection to achieve object transparency.

Acknowledgement

This work is supported by the National Science Foundation under grants CNS-0720505/0720242, CNS-0551658/0509186, and CCF-0540866/0540862, and by Microsoft, Intel, and IBM. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Bruneton, E., R. Lenglet and T. Coupaye, *ASM: a code manipulation tool to implement adaptable systems*, in: *Adaptable and Extensible Component Systems*, 2002.

[2] Caromel, D., W. Klauser and J. Vayssière, *Towards seamless computing and metacomputing in Java*, Concurrency—Practice and Experience **10** (1998), pp. 1043–1061.

[3] Chiba, S., *Load-time structural reflection in Java*, Lecture Notes in Computer Science **1850** (2000), p. 313.

[4] Chiba, S. and M. Nishizawa, *An easy-to-use toolkit for efficient Java bytecode translators*, in: *Proceedings of the International Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, 2003, pp. 364–376.

[5] Dahm, M., *Byte code engineering with the BCEL API*, Technical Report B-17-98, Freie Universität Berlin (2001).

[6] Hagimont, D. and D. Louvegnies, *Javanaise: distributed shared objects for Internet cooperative applications*, in: *Middleware'98*, The Lake District, England, 1998.

[7] Haumacher, B., J. Reuter and M. Philippsen, *JavaParty: A distributed companion to Java*, http://www.ipd.uka.de/JavaParty/.

[8] Huang, S. S. and Y. Smaragdakis, *Easy language extension with Meta-AspectJ*, in: *ICSE '06: Proceedings of the 28th international conference on Software engineering* (2006), pp. 865–868.

[9] Hunt, G. C. and M. L. Scott, *The Coign automatic distributed partitioning system*, in: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 1999, pp. 187–200.

[10] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science **2072** (2001), pp. 327–355.

[11] Launay, P. and J.-L. Pazat, *A framework for parallel programming in Java*, in: *HPCN Europe*, 1998, pp. 628–637.

[12] Launay, P. and J.-L. Pazat, *Generation of distributed parallel Java programs*, Technical Report PI-1171, Institut de Recherche en Informatique et Systemes Aleatoires (1998).

[13] Liogkas, N., B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich and S. Voida, *Automatic partitioning: A promising approach to prototyping ubiquitous computing applications* (2004).

[14] Messer, A., I. Greeberg, P. Bernadat and D. Milojicic, *Towards a distributed platform for resource-constrained devices* (2002).

[15] Philippsen, M. and M. Zenger, *JavaParty — transparent remote objects in Java*, Concurrency—Practice and Experience **9** (1997), pp. 1225–1242.

[16] Tatsubori, M., T. Sasaki, S. Chiba and K. Itano, *A bytecode translator for distributed execution of "legacy" Java software*, in: J. L. Knudsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science **2072** (2001), pp. 236–255.

[17] Terracotta Inc., *Terracotta*, http://terracotta.org.

[18] Tilevich, E. and Y. Smaragdakis, *J-Orchestra: Enhancing Java programs with distribution capabilities*, ACM Transactions on Software Engineering and Methodology To appear.

[19] Tilevich, E. and Y. Smaragdakis, *J-Orchestra: Automatic Java application partitioning*, in: B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science **2374** (2002), pp. 178–204.

[20] Tilevich, E. and Y. Smaragdakis, *Transparent program transformations in the presence of opaque code*, in: *Proceedings of the International Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, 2006.

Virtual-Machine Abstraction and Optimization Techniques

Stefan Brunthaler[1]

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 such machines. This paper presents an explanation 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, and ease of implementation. This also means that there is a huge 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 careful analyses and in-depth treatment of performance characteristics, implementation details.

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

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

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 sampled 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 bytecode 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],[20], 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.

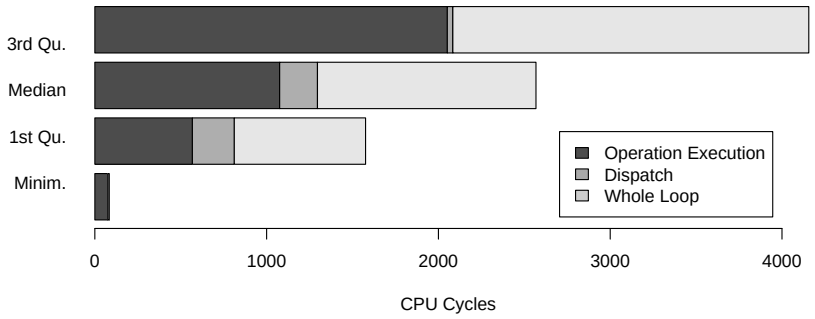


Fig. 1. CPU cycles per code section for Minimum, 1st quartile, median, and 3rd quartile measures.

Section	Min	1st Quartile	Median	3rd Quartile
Op-Execution	75	568	1076	2052
Dispatch	84	812	1296	2084
Whole Loop	84	1576	2568	4156

Table 1
Minimum, 1st quartile, median, and 3rd quartile values for CPU cycles per code section. All values here are *inclusive*, i.e., values for *Dispatch* include *Operation Execution*, and *Whole Loop* includes values of *Dispatch*, and *Operation Execution* respectively.

Our results do not support the assumption of instruction dispatch being the most costly operation for the Python interpreter, actually operation execution is. For a detailed explanation of why this is, we present a comparative example of one instruction implementation of the interpreters of Java, OCaml, Python, and Lua in Section 2.1:

Java, according to the latest Java Language Specification [10], the Java instruction set consists of 205 operations, including reserved opcodes. Instructions are typed for the following primitive types: integers, longs, floats, doubles, and addresses. In our example we take a look at the Sable VM, version 1.13.

OCaml, is a derivative of ML enriched with object oriented elements [13]. Version 3.11.0 contains 146 instructions. Among those are regular stack manipulation instructions, complemented by instructions to manipulate the environment, which is needed for function application, and evaluation respectively. Additionally, it contains direct support for integer operations, which are however not documented in the corresponding documentation [3].

Python, is a multi-paradigm dynamically typed programming language, that enables hybrid object-oriented/imperative programming with functional programming elements [18]. It has 93 instructions in Python 3.0rc1. Most of its operations support ad-hoc polymorphism, e.g. `BINARY_ADD` concatenates string operands, but does numerical addition on numerical ones [17].

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 instruction.

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

```
case SVM_INSTRUCTION_IADD:
    {
         jint value1 = stack[stack_size - 2].jint;
         jint value2 = stack[--stack_size].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):
    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

machine. Consequently, the virtual machine addition is expressed using a single native machine instruction. This constitutes our class of a *low abstraction level virtual machines*, where the interpreter is only a thin layer above a real machine.

```
BINARY_ADD:
    w = POP();
    v = TOP();
    if (PyUnicode_CheckExact(v) &&
        PyUnicode_CheckExact(w)) {
        x= unicode_concatenate(v, w, f, next_instr);
        goto skip_decref_vx;
    }
    else {
        x= PyNumber_Add(v, w);
    }
    Py_DECREF(v);
  skip_decref_vx:
    Py_DECREF(w);
    SET_TOP(x);
    if (x == NULL) continue;
    break;
```

Fig. 4. Implementation of integer addition in Python 3.0rc1.

Python's case (cf. Figure 4) shows a contrary picture: the upper arrow shows that `BINARY_ADD` does unicode string concatenation on string operands by calling `unicode_concatenate`. On non-string operands it calls `PyNumber_Add`, which implements dynamic typing and chooses the matching operation based on operand types, indicated by the lower arrow. In our integer example, the control flow would be: `PyNumber_Add`, `binary_op`, and finally `long_add`. If, however the operands were float, or complex types, then `binary_op` would have diverted to `float_add`, or `complex_add` respectively (cf. Figure 5).

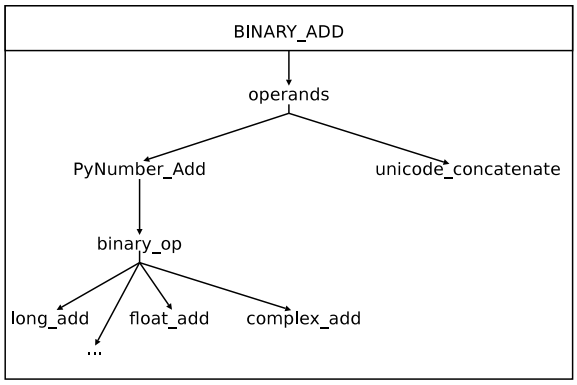


Fig. 5. Ad-hoc polymorphism in Python's `BINARY_ADD` instruction.

Aside from this ad-hoc polymorphism, the addition in Python 3.0 has an additional feature: it allows for unbounded range mathematics for integers, i.e., it is

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*.

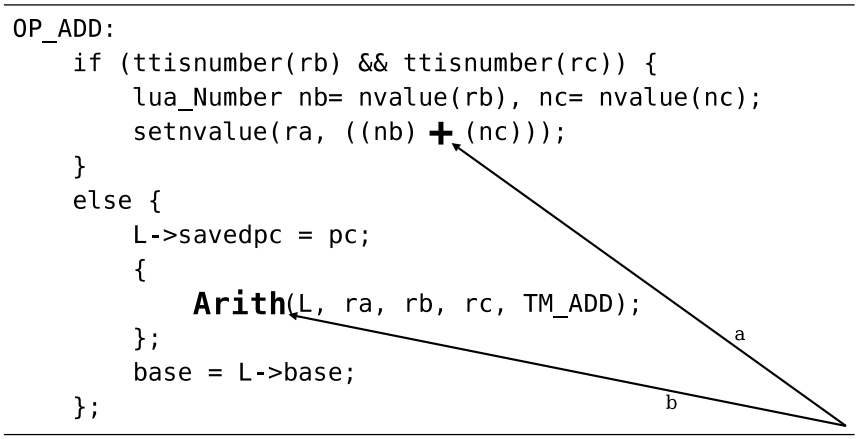


Fig. 6. Implementation of integer addition in Lua 5.1.4.

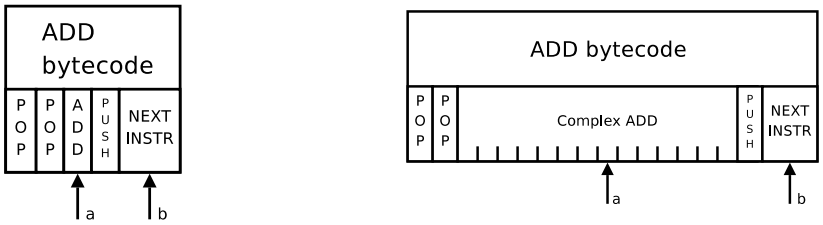
Our classes are by no means completely separated and disjoint, since interpreters can be members of both classes, having some subset of instructions belong into one set, and a separate subset of instructions to the other. This is exemplified in Lua (cf. Figure 6) in which addition has characteristics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

In Lua, if operand types are numeric it delegates the actual addition implementation to the compiler, and therefore to the machine (cf. Figure 6 arrow a). No distinction between float, double, long, and integers is necessary, because Lua uses double as its default numeric type. So far, this would indicate a low abstraction level. However, if operand types are non-numeric, Lua's implementation delegates to the `Arith` (cf. Figure 6, arrow b) function, which tries to convert these operand types into a numeric representation that can be added, e.g. if given a string operand which holds a non-ambiguous numerical value, it would extract this value and continue with regular addition. This "operand-polymorphism" is often found in other programming languages, too—e.g. in Perl—and constitutes a high abstraction level 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:



(a) Low Abstraction Level Virtual Machine

(b) High Abstraction Level Virtual Machine

Fig. 7. Illustration of Virtual Machine Abstraction Levels. Important are the different ratios of $a : b$ which affects the relative optimization potential of various optimization techniques.

- *Ad-Hoc Polymorphism*: a) either polymorphic operations are selected for concrete tuples 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.
- *Complex Operation Implementation*: In Python's case, this complexity directly maps to unbounded integer range mathematics for numeric operations, or full unicode 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

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

Table 2
Reference of reported speedup factors for several techniques.

techniques for optimizing the high abstraction level are implemented, the ratio of operation-execution vs. instruction-dispatch (as indicated by the arrows a and b in Figures 7(a) and 7(b)) has favorably changed their optimization potential. Therefore, our categorization merely provides an ordering of relative merits of various optimization techniques, such that considerable deviations in expected/documented vs. actually measured speedups are not stunningly surprising anymore.

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. Hölzle 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

match the cached pair (cf. Figure 8).

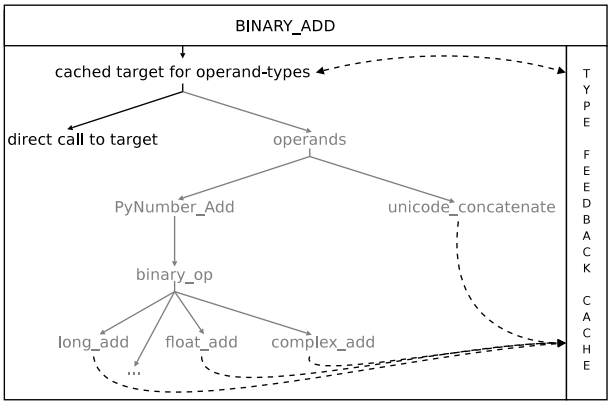


Fig. 8. Type feedback for Python's `BINARY_ADD` instruction. The gray-colored part is the system default look-up routine from Figure 5. The dashed arrows represent querying the type feedback cache (upper bidirectional arrow), and updating the cache with new target addresses after running through the 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 i 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, `IADD` and `LADD`, are bound by a maximum range of representable numbers—32-bit for integers, and 64-bit for long integers respectively—whereas Python's `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 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 `IADD`, or `LADD` for that matter, would be substituted by a invocation of a software algorithm for unbounded range mathematics of java.math.BigInteger—probably similar to the one implemented for Python's `BINARY_ADD`, or `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 `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 immanent 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, Hölzle 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, Hölzle 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

[1] "IVME '04: Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04)," ACM, New York, NY, USA, 2004, General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

[2] Bell, J. R., *Threaded code*, Communications of the ACM **16** (1973), pp. 370–372.

[3] Botlan, D. L. and A. Schmitt, "The OCaml System–Implementation," (2001), http://pauillac.inria.fr/~lebotlan/docaml_html/english/.

[4] CRACS, *Yap–Yet Another Prolog*, http://www.dcc.fc.up.pt/~vsc/Yap/.

[5] Ertl, M. A., *Threaded code variations and optimizations*, in: *EuroForth*, TU Wien, Vienna, Austria, 2001, pp. 49–55.

[6] Ertl, M. A. and D. Gregg, *The behavior of efficient virtual machine interpreters on modern architectures*, in: *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference on Parallel Processing* (2001), pp. 403–412.

[7] Ertl, M. A. and D. Gregg, *Optimizing indirect branch prediction accuracy in virtual machine interpreters*, in: *SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)* (2003), pp. 278–288.

[8] Ertl, M. A. and D. Gregg, *The structure and performance of efficient interpreters*, Journal of Instruction-Level Parallelism **5** (2003).

[9] Ertl, M. A. and D. Gregg, *Combining stack caching with dynamic superinstructions*, [1], pp. 7–14, General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

[10] Gosling, J., B. Joy, G. Steele and G. Bracha, "Java Language Specification, Second Edition: The Java Series," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] Hölzle, U., "Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming," Ph.D. thesis, Stanford, CA, USA (1995).

[12] Hölzle, U. and D. Ungar, *Optimizing dynamically-dispatched calls with run-time type feedback*, in: *SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, 1994, pp. 326–336.

[13] INRIA, *OCaml*, http://caml.inria.fr.

[14] Matsumoto, Y., *Ruby*, http://ruby-lang.org.

[15] Piumarta, I. and F. Riccardi, *Optimizing direct threaded code by selective inlining*, in: *SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)* (1998), pp. 291–300.

[16] PUC-Rio, *Lua*, http://www.lua.org.

[17] Python Software Foundation, *Python*, http://www.python.org.

[18] python.org, "Python v3.0 documentation," (2008), http://docs.python.org/3.0/.

[19] Romer, T. H., D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad and H. M. Levy, *The structure and performance of interpreters*, in: *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (1996), pp. 150–159.

[20] Shi, Y., K. Casey, M. A. Ertl and D. Gregg, *Virtual machine showdown: Stack versus registers*, ACM Transactions on Architecture and Code Optimization **4** (2008), pp. 1–36.

[21] The Perl Foundation, *Perl*, http://www.perl.org.

[22] Vitale, B. and T. S. Abdelrahman, *Catenation and specialization for Tcl virtual machine performance*, [1], pp. 42–50, General Chair-Michael Franz and Program Chair-Etienne M. Gagnon.

Towards an XML-based Bytecode Level Transformation Framework

Arno Puder[1] and Jessica Lee[2]

San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132

Abstract

Virtual machines (VMs) facilitate the deployment of applications in heterogeneous environments. Popular VMs such as Sun Microsystem'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, supporting technologies such as XSL stylesheets 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] Email: arno@sfsu.edu
[2] Email: jeslee@sfsu.edu

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

Programs to manipulate virtual machine bytecode are becoming more prevalent. BAT2XML [3] 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 authorization. These concerns are shared by modules all 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 byte code 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 to run on a JVM. However, neither of these approaches can provide a generic representation of bytecode for both the CLR and the JVM.

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

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"
       xmlns="http://xmlvm.org">
    <class ...>
        <field .../>
        <method ...>
            <signature>...</signature>
            <code>...</code>
        </method>
    </class>
</xmlvm>
```

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:

```
// C#
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

simplified representation of class `AddTest` in XMLVM:

```
<xmlvm xmlns:clr="http://xmlvm.org/clr"
       xmlns="http://xmlvm.org">
  <class name="AddTest">
    <method name="Main" isStatic="true" isPublic="true">
      <signature>
        <return type="void" />
      </signature>
      <code>
        <clr:var index="0" type="int" />
        <clr:var index="1" type="int" />
        <clr:ldc type="int" value="11" />
        <clr:stloc index="0" />
        <clr:ldc type="int" value="22" />
        <clr:stloc index="1" />
        <clr:ldloc index="0" />
        <clr:ldloc index="1" />
        <clr:add />
        <clr:call has-this="false" class-type="System.Console" method="WriteLine">
          <signature>
            <return type="void" />
            <parameter type="int" />
          </signature>
        </clr:call>
        <clr:return />
      </code>
    </method>
  </class>
</xmlvm>
```

It should be emphasized again that the above XMLVM program is essentially an XML representation of the contents of the `AddTest.exe` executable generated by a C# compiler. The top-level tags are identical to the XML template shown earlier. The <`method`> tag has attributes for each of the modifiers `public` and `static`. A method has access to its own stack and local variables as well as the global heap. If a method has actual parameters, they are automatically stored in the local variables upon entering the method.

The most interesting part of the above XMLVM program is the actual implementation of method `Main`, which lies in between the tags <`code`> and </`code`>. Since the bytecode instructions belong to the CLR, the respective XMLVM instructions are placed in the XML namespace denoted by the prefix `clr`. The <`clr:var`> (*variable*) tag declares a variable with respective type that can be addressed by a given index. Instruction <`clr:ldc`> (*load constant*) pushes a constant referred to by attribute `value` onto the stack.

The <`clr:stloc`> (*store location*) instruction pops off the top of the stack and saves it in the local variable referred to by attribute `index`. The <`clr:ldloc`> (*load location*) instruction does the inverse by pushing the content of a variable onto the stack. The instruction <`clr:add`> (*addition*) pops the last two values off the stack and pushes their sum back onto the stack. The <`clr:call`> instruction invokes the method `System.Console.WriteLine()`. The `false` value of attribute `has-this` indicates that `WriteLine()` is a static method, because it does not require a `this` reference. Note that the actual parameters have been removed by the <`clr:call`> instruction after the invocation. The <`clr:return`> instruction finally leaves the method `Main` and returns to the caller.

The aforementioned <`clr:add`> instruction gives no indication as to the type of the operands. In this particular example, the <`clr:add`> instruction will compute the sum of two integers, since the two top elements of the stack are of type integer.

The same <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 <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 `checked((int) (a + b))`, it would have resulted in the byte code instruction <clr:add_ovf> (*add overflow*) that raises a runtime exception when an overflow occurs. Note that for the <clr:add_ovf> instruction the VM also has to determine the correct type of the arguments similar to <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 how both the CLR's type-agnostic instructions (as used with primitive types) and value types can be mapped to the JVM. Other features such as delegates and generics are left for future work. The work involved with this mapping will allow us to demonstrate the power and flexibility of using XML, in conjunction with XSL and XPath, to perform cross-compilation. The features discussed in the following already allow for 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

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 signed 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 low-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.

Whereas 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_ovf>: adds two signed integer values with overflow check. Throws `OverflowException` if an overflow occurs.
- <clr:add_ovf_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_ovf_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.

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:

```
1 // C#
2 using System;
3
4 public struct Person {
5     public string Name;
6
7     public Person(string name) {
8         Name = name;
9     }
10 }
11
12 class ValueTypeTest {
13     static void Main() {
14         Person p1 = new Person("Bob");
15         Person p2 = p1;
16         p2.Name = "Alice";
17         assert(p1.Name == "Bob");
18         assert(!p1.Equals(p2));
19     }
20 }
```

In C#, a `struct` defines a value type (line 4). Although a value type is also instantiated via the `new` operator (line 14), it will effectively be allocated on the stack. As a consequence, variable `p2` in the example above will copy the value type (line 15). If `Person` were a proper class, `p1` and `p2` would be referencing the same object. But as can be seen by the assertions in lines 17 and 18, `p1` and `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

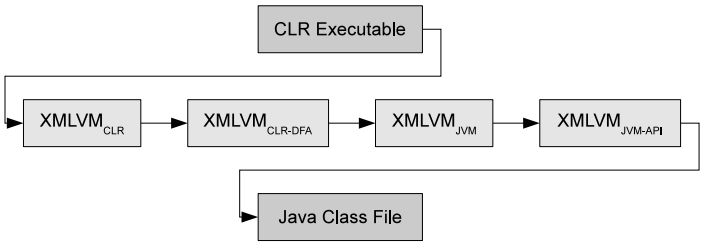


Fig. 1. XMLVM Toolchain.

technologies. The basis for this study is the use of XMLVM, which was introduced in Section 2. Figure 1 summarizes the overall workflow of our toolchain. The transformation begins with a CLR executable that is first translated to an XML representation we call XMLVM$_{CLR}$, whose format was described earlier. The next step consists of a data flow analysis of the XMLVM program which we refer to as XMLVM$_{CLR-DFA}$. Section 4.1 details how the data flow analysis can be done via XSL stylesheets. After the data flow analysis, the CLR bytecode instructions can be transformed to the JVM (referred to XMLVM$_{JVM}$ in Figure 1), the details of which will be given in Section 4.2. After cross-compiling the bytecode instructions, the API of external libraries need to be mapped which is described in Section 4.3. Finally, the resulting XMLVM$_{JVM-API}$ program can be translated to a binary Java class file that can be executed on a standard JVM.

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`>

tag. We refer to the resulting XMLVM program that contains these type-stack annotations as XMLVM$_{CLR-DFA}$.

We use XSL templates to generate the type stack transition for each instruction. XSL templates specify translations to apply to an XML document when an XML node is found that matches the rule specified in the `match` attribute. As a specific example, consider the aforementioned <`clr:ldloc`> instruction that pushes the content of a local variable onto the stack. In terms of changes to the type stack, this instruction will push an argument of a certain type (determined by the type of the variable whose value is to be pushed onto the stack). The following is an example of the XSL template for <`clr:ldloc`>:

```
1 <xsl:template match="clr:ldloc[preceding-sibling::*[1]/dfa:stack-post]">
2     <clr:ldloc>
3         <xsl:copy-of select="@*"/>
4         <stack-pre>
5             <xsl:copy-of select="preceding-sibling::*[1]/stack-post/*"/>
6         </stack-pre>
7         <stack-post>
8             <xsl:copy-of select="preceding-sibling::*[1]/stack-post/*"/>
9             <elem>
10                <xsl:variable name="idx" select="@index"/>
11                <xsl:attribute name="type">
12                    <xsl:value-of select="../clr:var[@index = $idx]/@type"/>
13                </xsl:attribute>
14            </elem>
15        </stack-post>
16    </clr:ldloc>
17 </xsl:template>
```

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 `match` attribute (line 1). The first <`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 <`stack-pre`> tag. The state of the type stack at this point is identical with the <`stack-post`> of the previous instruction. Therefore, the XPath expression of the second <`copy-of`> selects the <`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 <`clr:ldloc`> instruction via the <`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 pushed by <`clr:ldloc`>. The <`value-of`> in the template above (line 12) uses an XPath expression to reference that type. The following excerpt shows the resulting XMLVM$_{CLR-DFA}$ program for `AddTest` with the computed type stack transitions:

```
1 <!-- ... -->
2 <clr:ldloc index="0">
3     <stack-pre/>
4     <stack-post>
5         <elem type="int"/>
6     </stack-post>
7 </clr:ldloc>
8 <clr:ldloc index="1">
9     <stack-pre>
10        <elem type="int"/>
11    </stack-pre>
12    <stack-post>
13        <elem type="int"/>
14        <elem type="int"/>
15    </stack-post>
16 </clr:ldloc>
```

```
17 <clr:add>
18     <stack-pre>
19         <elem type="int"/>
20         <elem type="int"/>
21     </stack-pre>
22     <stack-post>
23         <elem type="int"/>
24     </stack-post>
25 </clr:add>
26 <!-- ... -->
```

In the XMLVM$_{CLR-DFA}$ 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

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:ldnull`> 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$_{CLR}$ program needs to be mapped to one of the typed addition operators of the JVM. This can be accomplished with the help of the data flow analysis. The following two XSL templates demonstrate the mapping of the CLR instruction <`clr:add`> for integer types:

```
1 <xsl:template match="clr:add[stack-post/elem[last()][@type = 'int']]">
2     <jvm:iadd/>
3 </xsl:template>
4
5 <xsl:template match="clr:add[stack-post/elem[last()][@type = 'long']]">
6     <jvm:ladd/>
7 </xsl:template>
```

The XPath expression in the `match`-statements of the XSL templates check the top of the type stack and map the <`clr:add`> CLR instruction to either the JVM <`jvm:iadd`> instruction (line 2) or the JVM <`jvm:ladd`> instruction (line 6) depending whether the top of the type stack is of type `int` or `long`. This is an example of how un-typed CLR instructions can be mapped to type-specific JVM instructions via declarative XSL templates. The following XMLVM excerpt shows the JVM bytecode instructions that result from transforming the CLR implemen-

tation of the `AddTest` class introduced earlier:

```
<code>
    <jvm:ldc type="int" value="11"/>
    <jvm:istore index="0"/>
    <jvm:ldc type="int" value="22"/>
    <jvm:istore index="1"/>
    <jvm:iload index="0"/>
    <jvm:iload index="1"/>
    <jvm:iadd/>
    <jvm:invokestatic class-type="System.Console" method="WriteLine">
        <signature>
            <return type="void"/>
            <parameter type="int"/>
        </signature>
    </jvm:invokestatic>
    <jvm:return/>
</code>
```

In some cases it is not possible to map one CLR bytecode instruction to exactly one JVM instruction. The instructions that check for overflow (both <clr:add_ovf> and <clr:add_ovf_un>) during addition are two such cases. In order to map those CLR instructions to the JVM, it is necessary to introduce a compatibility library that mimics the semantics of the original CLR instruction. As an example, consider the CLR instruction <clr:add_ovf> discussed earlier. Since the JVM does not offer a single bytecode instruction with the same semantics, the <clr:add_ovf> instruction is mapped via the following stylesheet to an invocation of the static method `MathLib.add_ovf(int, int)`:

```
<xsl:template match="clr:add_ovf[stack-post/elem[last()][@type = 'int']]">
    <jvm:invokestatic class-type="MathLib" method="add_ovf">
        <signature>
            <return type="int"/>
            <parameter type="int"/>
            <parameter type="int"/>
        </signature>
    </jvm:invokestatic>
</xsl:template>
```

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:

```
// Java
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();
    }
}
```

4.2.2 Mapping Value Types

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

```
1 <clr:var index="0" isValueType="true" type="Person" />
2 <clr:var index="1" isValueType="true" type="Person" />
3 <clr:ldloca index="0" />
4 <clr:ldc type="String" value="Bob" />
5 <clr:call has-this="true" class-type="Person" method=".ctor">
6     <vm:signature>
7         <vm:return type="void" />
8         <vm:parameter type="String" />
9     </vm:signature>
10 </clr:call>
11 <clr:ldloc index="0" />
12 <clr:stloc index="1" />
```

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:ldloca> (*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 byte code 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 `__COPY` to this class that performs the deep copy using the Java reflection API. Here is the declaration of this method:

```
1 // Java
2 package System;
3
4 public class ValueType extends System.Object {
```

```
5     static public void __COPY(ValueType from, ValueType to)
6     { ... }
7 }
```

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:

```
1 <jvm:aload index="1" type="Person"/>
2 <jvm:invokestatic class-type="System.ValueType" method="__COPY">
3     <vm:signature>
4         <vm:return type="void"/>
5         <vm:parameter type="System.ValueType"/>
6         <vm:parameter type="System.ValueType"/>
7     </vm:signature>
8 </jvm:invokestatic>
```

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 WinForms 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. Class `System.String` in .NET defines the read-only property `Length` for that purpose:

```
1 // C#
2 namespace System {
3     class String {
4         public int Length {get;}
5         // ...
6     }
7 }
```

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:

```
<xsl:template match="jvm:invokevirtual[@class-type = 'System.String' and
                                       @method = 'get_Length']">
    <jvm:invokevirtual class-type="java.lang.String" method="length">
        <vm:signature>
            <vm:return type="int"/>
        </vm:signature>
    </jvm:invokevirtual>
</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 capabilities of XMLVM to represent bytecode programs that use instructions from different VMs. In particular, we plan to look at the possibility of weaving byte code instructions from different VMs into one XMLVM program in the context of AOP. This would make it possible to weave a C# aspect into a Java application and vice-versa. The cross-compilation would have to cope with mixed bytecode instructions from different VMs. XMLVM would be particularly well-suited to serve as an abstraction for this heterogeneous mix of bytecode instructions.

References

[1] Dahm, M., *Byte code engineering*, Java Informations Tage, 1999, pp. 267–277.

[2] ECMA, "Common Language Infrastructure (CLI)," 4th edition (2006).

[3] Eichberg, M., *BAT2XML: XML-based Java Bytecode Representation*, Electronic Notes in Theoretical Computer Science, 2005.

[4] Frijters, J., "IKVM.NET: A JVM for the Microsoft .NET Framework," Http://www.ikvm.net.

[5] Goo, A., "JaCIL: A CLI to JVM Compiler," Http://sourceforge.net/projects/jacil/.

[6] Gosling, J., *Java intermediate bytecodes*, Proc. ACM SIGPLAN Workshop on Intermediate Representations (1995), pp. 111–118.

[7] Laddad, R., "AspectJ in Action: Practical Aspect-Oriented Programming," Manning Publications, 2003.

[8] Leroy, X., *Java bytecode verification: algorithms and formalizations*, Journal of Automated Reasoning **30** (2003), pp. 235–269.

[9] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley Pub Co, 1999, second edition.

[10] W3C, "XSL Transformations," (1999), http://www.w3.org/TR/xslt.

[11] W3C, "eXtensible Markup Language (XML)," (2006), http://www.w3.org/XML/.

The S3MS.NET Run Time Monitor

Lieven Desmet[1] Wouter Joosen[1] Fabio Massacci[2]
Katsiaryna Naliuka[2] Pieter Philippaerts[1,3] Frank Piessens[1]
Dries Vanoverberghe[1]

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.
This paper describes the overall functionality and architecture of the tool, discusses its strengths and weaknesses, and reports on our experience with using the tool on case studies as well as in teaching.

Keywords: security, bytecode rewriting, .NET, MSIL

1 Introduction

In todays 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] DistriNet Research Group, Department of Computer Science Katholieke Universiteit Leuven, Celestijnlaan 200A, B-3001 Leuven, Belgium
[2] Department of Information and Communication Technology, Universit di Trento, Via Sommarive 14, I-38050 Povo (Trento), Italy
[3] Email: Pieter.Philippaerts@cs.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 S3MS.NET run time monitor consists of two key components (Figure 1). The *inliner* 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.

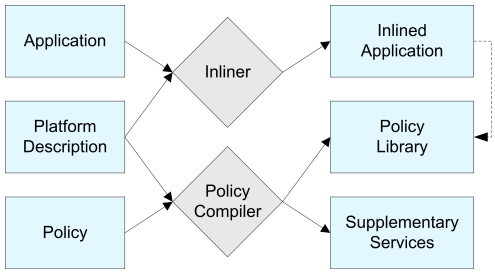


Fig. 1. The architecture of the tool

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 Vanoverberghe and Piessens [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. If the method to send SMS's is considered security relevant, the inliner will transform it as shown. Note that the tool operates on the level of bytecode, not on source level, but we show the results as they would look at source level to make the transformation easier to understand.

```
SmsMessage message = ...
message.SendSMS();
```

Listing 1. Example code that sends an SMS message on a mobile phone.

```
SmsMessage message = ...
PDP.BeforeSendSMS(message);
try {
    message.SendSMS();
    PDP.AfterSendSMS(message);
} catch (SecurityException se) {
    throw se;
} catch (Exception e) {
    PDP.ExceptionSendSMS(message, e);
    throw;
}
```

Listing 2. The SMS example code, after inlining.

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 the 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 PDP 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 security state, 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 ([1]) 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

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_G) points to the final state of the previous session.

3 Experience and Discussion

The S3MS.NET run time monitor was developed in the European FP6 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

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 [9] show that the performance impact of the inlining itself is negligible, and for the policies and case studies done in S3MS, 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 2D-LTL.

References

[1] Aktug, I. and K. Naliuka, *Conspec - a formal language for policy specification*, Electr. Notes Theor. Comput. Sci. **197** (2008), pp. 45–58.

[2] Desmet, L., W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens and D. Vanoverberghe, *A flexible security architecture to support third-party applications on mobile devices*, in: *CSAW*, 2007, pp. 19–28.

[3] Desmet, L., W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan and D. Vanoverberghe, *Security-by-contract on the .NET platform*, Inf. Secur. Tech. Rep. **13** (2008), pp. 25–32.

[4] Edjlali, G., A. Acharya and V. Chaudhary, *History-based access control for mobile code*, in: *Proceedings of the 5th ACM conference on Computer and communications security*, 1998, pp. 38–40.

[5] Massacci, F. and K. Naliuka, *Multi-session security monitoring for mobile code*, Technical Report DIT-06-067, UNITN (2006).

[6] Schneider, F. B., *Enforceable security policies*, ACM Trans. Inf. Syst. Secur. **3** (2000), pp. 30–50.

[7] Úlfar Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. thesis, Dep. of Computer Science, Cornell University (2004).

[8] Vanoverberghe, D. and F. Piessens, *A caller-side inline reference monitor for an object-oriented intermediate language*, in: *FMOODS*, 2008, pp. 240–258.

[9] Vanoverberghe, D. and F. Piessens, *Security enforcement aware software development*, Information and Software Technology (2009).

[10] Zobel, A., C. Simoni, D. Piazza, X. Nuez and D. Rodriguez, *Business case and security requirements*, Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, Report available at www.s3ms.org (2006).

Soundly Handling Static Fields: Issues, Semantics and Analysis

Laurent Hubert[a,1,2] David Pichardie[b,1]

[a] *CNRS, IRISA, Campus Beaulieu , F-35042 Rennes Cedex, France*

[b] *INRIA, Centre Rennes — Bretagne Atlantique*
IRISA, Campus Beaulieu , F-35042 Rennes Cedex, France

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 compute, for each program point, the set of static fields that must have been initialized and discuss its 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 preserving 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 followings.

[1] Email:`first.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`

- 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 null-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 the class initializers it not intuitive and give 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 `A.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*,

```
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`.

```
class A extends Object{
  public static int CST= B.SIZE;}
class B extends Object{
  public static int SIZE = A.CST+5;
}
```

Fig. 2. Integer initial values can also depend on foreign code

it is therefore not decidable in general.

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 `B.<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.<clinit>`, 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`.

```
class A extends Object{
  static EMPTY=new A("");
  static ALL=new HashMap();

  String name;
  public A(String name){
    this.name = name;
    ALL.add(name,this);
  }
}
```

(a) An uninitialized field read leads to a `NullPointerException`

```
class A extends Object{
  static ALL=new HashMap();
  static EMPTY=new A("");

  String name;
  public A(String name){
    this.name = name;
    ALL.add(name,this);
  }
}
```

(b) No uninitialized field is read

Fig. 3. The issue can arise with a single class

(b) is the correct version, where the initializations of `ALL` and `EMPTY` have been switched. This example is an extract of `java.lang.Character$UnicodeBlock` of Sun's Java Runtime Environment (JRE) that we have simplified: we want the analysis to handle such cases. If we consider that `A` depend on itself, then we forbid way to much programs. If we do not consider that `A` depend on itself and we do not reject (a) then the analysis is incorrect. We therefore cannot rely on circular dependencies between classes.

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

- `put`(f) updates a field $f \in \mathbb{F}$.
- `invoke` calls a method (we do not mention the name of the target method because it would be redundant with the $\mathit{flow}_{\text{inter}}$ information described below).
- `return` returns 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.

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 $(m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$ where:

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

and such that *instr* and $\mathit{flow}_{\text{intra}}$ satisfy the following property:

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

In practice, $\mathit{flow}_{\text{clinit}}$ will contain all the pairs (l, C) of a bytecode program such that the instruction found at program point l is of the form `new` C, `putstatic` $C.f$, `getstatic` $C.f$ or `invokestatic` $C.m$ (see Section 2.17.4 of [12]). [3]

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, `A` is not initialized but `B` is. `A.<clinit>` is potentially called at exit point 8 but since 8 is only reachable after a first call to `A.<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.

$$
\begin{aligned}
v \in \text{Value} \quad & (\textit{abstract}) \\
s \in \text{Static} \quad & \stackrel{\text{def}}{=} \mathbb{F} \to \text{Value} + \{\Omega\} \\
h \in \text{History} \quad & \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{C}) \\
cs \in \text{Callstack} \quad & \stackrel{\text{def}}{=} (\{0,1\} \times \mathbb{P})^\star \\
<l, cs, s, h> \in \text{State} \quad & \stackrel{\text{def}}{=} \mathbb{P} \times \text{Callstack} \times \text{Static} \times \text{History}
\end{aligned}
$$

[3] 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.

$$\frac{\mathit{NeedInit}(l, C, h)}{<l, cs, s, h> \to <C.\texttt{<clinit>}.\mathit{first}, (\!|l|\!) :: cs, s, h \cup \{C\}>}$$

$$\frac{<l, cs, s, h> \to_1 <l', cs', s', h'> \qquad \forall C,\ \neg \mathit{NeedInit}(l, C, h)}{<l, cs, s, h> \to <l', cs', s', h'>}$$

$$\frac{\mathit{instr}(l) = \texttt{put}(f) \qquad \mathit{flow}_{\text{intra}}(l, l') \qquad v \in \text{Value}}{<l, cs, s, h> \to_1 <l', cs, s[f \mapsto v], h>}$$

$$\frac{\mathit{instr}(l) = \texttt{any} \qquad \mathit{flow}_{\text{intra}}(l, l')}{<l, cs, s, h> \to_1 <l', cs, s, h>} \qquad \frac{\mathit{instr}(l) = \texttt{invoke} \qquad \mathit{flow}_{\text{inter}}(l, m)}{<l, cs, s, h> \to_1 <m.\mathit{first}, l :: cs, s, h>}$$

$$\frac{\mathit{instr}(l) = \texttt{return} \quad \mathit{flow}_{\text{intra}}(l', l'')}{<l, l' :: cs, s, h> \to_1 <l'', cs, s, h>} \qquad \frac{\mathit{instr}(l) = \texttt{return} \quad <l', cs, s, h> \to_1 \ st'}{<l, (\!|l'|\!) :: cs, s, h> \to_1 \ st'}$$

Fig. 4. Operational semantics

A field contains either a value or a default value represented by the symbol Ω. Since a class initializer cannot be called twice in a same execution, we need to remember the set of classes whose initialization has been started (but not necessarily ended). This is the purpose of the element $h \in \text{History}$. Our language is given a small-step operational semantics with states of the form $<l, cs, s, h>$, where the label l uniquely identifies the current program point, cs is a call stack that keeps track of the program points where method calls have occurred, s associates to each field its value or Ω and h is the history of class initializer calls. Each program point l of a call stack is tagged with a Boolean which indicates whether the call in l was a call to a class initializer (the element of the stack is then noted $(\!|l|\!)$) or was a standard method call (simply noted l).

The small-step relation $\to \subseteq \text{State} \times \text{State}$ is given in Fig. 4 (we left implicit the program $(m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$ that we consider). It is based on the relation $\mathit{NeedInit} \subseteq \mathbb{P} \times \mathbb{C} \times \text{History}$ defined by

$$\mathit{NeedInit}(l, C, h) \quad \stackrel{\text{def}}{=} \quad \mathit{flow}_{\text{clinit}}(l) = C \ \wedge \ C \notin h$$

which means that the class initializer of class C must be called at program point l if and only if there is a corresponding edge in $\mathit{flow}_{\text{clinit}}$ and $C.\texttt{<clinit>}$ has not been called yet (*i.e.* $C \notin h$).

The relation $\to$ is defined by two rules. In the first one, the class initializer of class C needs to be called. We hence jump to the first point of $C.\texttt{<clinit>}$, push on the call stack the previous program point (marked with the flag $(\!|\cdot|\!)$) and record C in the history h. In the second rule, there is no need to initialize a class, hence we simply use the standard semantic of the current instruction, given by the relation $\to_1 \subseteq \text{State} \times \text{State}$.

The relation $\to_1$ is defined by five rules. The first one corresponds to a field update $\texttt{put}(f)$: an arbitrary value v is stored in field f. The second rule illustrates that the instruction `any` does not affect the visible elements of the state. For a method call (third rule), the current point is pushed on the call stack and the

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 Ω.

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

$$[\![p]\!] = \{ <i, cs, s, h> \mid <m_0.\mathit{first}, \varepsilon, \lambda f.\Omega, \emptyset> \rightarrow^\star <i, cs, s, h> \}$$

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 *Wf* 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 $\mathit{flow}_{\text{clinit}}(l) = C$, two exclusive cases may happen:

(i) C.`<clinit>` has not been called yet: C.`<clinit>` is immediately called.

(ii) C.`<clinit>` has already been called (and may still be in progress): C.`<clinit>` will not be called a second time.

Using the initialization information given by C.`<clinit>` is safe only in case (i), *i.e.* the control flow information in $\mathit{flow}_{\text{clinit}}$ 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 analysis 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 initializer of `B`. At the beginning of the class initializer of `B`, *May* equals to $\{\texttt{B}, \texttt{C}\}$. We compute the set of fields initialized by `A.<clinit>`, which is $\{\texttt{A.f}\}$. As `A` is not in *May* at the beginning of `B.<clinit>`, we can assume the class initializer 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 $\{\texttt{A}, \texttt{B}, \texttt{C}\}$, which makes impossible to assume anymore that `A.<clinit>` is called 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 $\texttt{C} \in \mathit{Must}$ 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 $\mathit{flow}_{\text{clinit}}$. 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 $p = (m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$. We note $\mathcal{P}_p(\mathbb{C})$ (resp. $\mathcal{P}_p(\mathbb{F})$) the finite set of classes (reps. fields) that appears in p. For each program point $l \in \mathbb{P}$, we compute before and after the current point three sets of data $(\mathit{May}, \mathit{Must}, \mathit{Wf}) \in \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{F})$. 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 $(A^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ where:

- $A^\sharp = \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{F})$.

- $\bot = (\emptyset, \mathcal{P}_p(\mathbb{C}), \mathcal{P}_p(\mathbb{F}))$.
- $\top = (\mathcal{P}_p(\mathbb{C}), \emptyset, \emptyset)$.
- for all $(May_1, Must_1, Wf_1)$ and $(May_2, Must_2, Wf_2)$ in $A^\sharp$,

$$(May_1, Must_1, Wf_1) \sqsubseteq (May_2, Must_2, Wf_2) \text{ iff}$$
$$May_1 \subseteq May_2,\ Must_1 \supseteq Must_2 \text{ and } Wf_1 \supseteq Wf_2$$
$$(May_1, Must_1, Wf_1) \sqcup (May_2, Must_2, Wf_2) =$$
$$(May_1 \cup May_2, Must_1 \cap Must_2, Wf_1 \cap Wf_2)$$
$$(May_1, Must_1, Wf_1) \sqcap (May_2, Must_2, Wf_2) =$$
$$(May_1 \cap May_2, Must_1 \cup Must_2, Wf_1 \cup Wf_2)$$

Each element in $A^\sharp$ expresses properties on fields and on an initialization historic. This is formalized by the following correctness relation.

Definition 4.2 [Correctness relation] $(May, Must, Wf)$ is a correct approximation of $(s, h) \in \text{Static} \times \text{History}$, written $(May, Must, Wf) \sim (s, h)$ iff:

$$Must \subseteq h \subseteq May \quad \text{and} \quad Wf \subseteq \{\ f \in \mathbb{F} \mid s(f) \neq \Omega\ \}$$

This relation expresses that

(i) *May* contains all the classes for which we may have called the `<clinit>` method since the beginning of the program (but it may not be finished yet).

(ii) *Must* contains all the classes for which we must have called the `<clinit>` method since the beginning of the program (but it may not be finished yet neither).

(iii) *Wf* contains all the fields for which we are sure they have been written at least once.

The analysis is then specified as a data flow problem.

Definition 4.3 [Data flow solution] A *Data flow solution* of the Must-Have-Been-Initialized analysis is any couple of maps $A_{\text{in}}, A_{\text{out}} \in \mathbb{P} \to A^\sharp$ that satisfies the set of equations presented in Fig. 6, for all program point l of the program p.

In this equation system, $A_{\text{in}}(l)$ is the abstract union of three kinds of data flow information: (i) $A_0(l)$ gives the abstraction of the initial states if l is the starting point of the main method m_0; (ii) $A_{first}(l)$ is the abstract union of all calling contexts that may be transferred to l if it is the starting point of a method m. We distinguish two cases, depending on whether m is the class initializer of a class C. If it is, incoming calling contexts are transformed with $F^{\text{init}}_{\text{call}}$ which filters unfeasible calling edges with *Must* and adds C to *May* and *Must* otherwise. Otherwise, incoming calling contexts are transformed with F^{init} (described below) which take into account the potential class initialization that may have been performed before the call. (iii) At last, we merge all incoming data flows from predecessors in the intra-procedural graph.

$$A_{\text{in}}(l) = A_0(l) \sqcup A_{\mathit{first}}(l) \sqcup \bigsqcup\{A_{\text{out}}(l') \mid \mathit{flow}_{\text{intra}}(l', l)\}$$

$$A_{\text{out}}(l) = \begin{cases} F_{\text{call}}\left(F^{\text{init}}(l, A_{\text{in}}(l)), \bigsqcup\{A_{\text{in}}(m.\mathit{last}) \mid \mathit{flow}_{\text{inter}}(l, m)\}\right) & \text{if } \mathit{instr}(l) = \texttt{invoke} \\ F_{\mathit{instr}(l)}(F^{\text{init}}(l, A_{\text{in}}(l))) & \text{otherwise} \end{cases}$$

where

$$A_0(l) = \begin{cases} (\emptyset, \emptyset, \emptyset) & \text{if } l = m_0.\mathit{first} \\ \bot & \text{otherwise} \end{cases}$$

$$A_{\mathit{first}}(l) = \begin{cases} \bigsqcup\left\{F^{\text{init}}_{\text{call}}(C, A_{\text{in}}(l')) \mid \mathit{flow}_{\text{clinit}}(l') = C\right\} & \text{if } l = C.\texttt{<clinit>}.\mathit{first} \\ \bigsqcup\left\{F^{\text{init}}(l', A_{\text{in}}(l')) \mid \mathit{flow}_{\text{inter}}(l', m)\right\} & \text{if } l = m.\mathit{first} \\ \bot & \text{otherwise} \end{cases}$$

and $F_{\texttt{return}}$, $F_{\texttt{any}}$, $F_{\texttt{put}(f)} \in A^\sharp \to A^\sharp$, $F_{\text{call}} \in A^\sharp \times A^\sharp \to A^\sharp$, $F^{\text{init}}_{\text{call}} \in \mathbb{C} \times A^\sharp \to A^\sharp$ and $F^{\text{init}} \in \mathbb{P} \times A^\sharp \to A^\sharp$ are transfer functions defined by:

$$F_{\texttt{return}}(\mathit{May}, \mathit{Must}, \mathit{Wf}) = F_{\texttt{any}}(\mathit{May}, \mathit{Must}, \mathit{Wf}) = (\mathit{May}, \mathit{Must}, \mathit{Wf})$$
$$F_{\texttt{put}(f)}(\mathit{May}, \mathit{Must}, \mathit{Wf}) = (\mathit{May}, \mathit{Must}, \mathit{Wf} \cup \{f\})$$

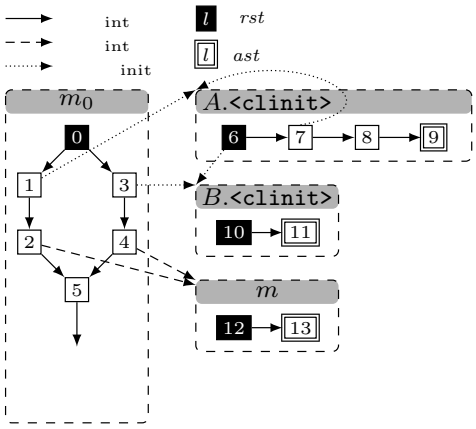
$$F_{\text{call}}((\mathit{May}_1, \mathit{Must}_1, \mathit{Wf}_1), (\mathit{May}_2, \mathit{Must}_2, \mathit{Wf}_2)) = (\mathit{May}_2, \mathit{Must}_1 \cup \mathit{Must}_2, \mathit{Wf}_1 \cup \mathit{Wf}_2)$$

$$F^{\text{init}}_{\text{call}}(C, (\mathit{May}, \mathit{Must}, \mathit{Wf})) = \begin{cases} \bot & \text{if } C \in \mathit{Must} \\ (\mathit{May} \cup \{C\}, \mathit{Must} \cup \{C\}, \mathit{Wf}) & \text{otherwise} \end{cases}$$

$$F^{\text{init}}(l, a) = \begin{cases} F_{\text{call}}(a, A_{\text{in}}(C.\texttt{<clinit>}.\mathit{last})) & \text{if } \mathit{flow}_{\text{clinit}}(l) = C \text{ and } C \notin \mathit{May} \\ a & \text{if } (\mathit{flow}_{\text{clinit}}(l) = C \text{ and } C \in \mathit{Must}) \\ & \text{ or } \forall C.(\mathit{flow}_{\text{clinit}}(l) \neq C) \\ a \sqcup F_{\text{call}}(a, A_{\text{in}}(C.\texttt{<clinit>}.\mathit{last})) & \text{otherwise} \end{cases}$$

Fig. 6. Data flow analysis

The equation on $A_{\text{out}}(l)$ distinguishes two cases, depending on $\mathit{instr}(l)$ 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_{call} described below, with the data flows facts $F^{\text{init}}(l, A_{\text{in}}(l))$ that is true just before the call. Otherwise, we transfer the data flow $A_{\text{in}}(l)$, found at entry of the current instruction with F^{init} and then $F_{\mathit{instr}(l)}$. While F^{init} handles potential class initialization that may have been performed before the instruction, $F_{\mathit{instr}(l)}$ simply handles the effect of the instruction



(a) A program example

l	$instr(l)$	$A_{\text{in}}(l)$			$A_{\text{out}}(l)$		
		May	*Must*	*Wf*	*May*	*Must*	*Wf*
0	any	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	any	$\emptyset$	$\emptyset$	$\emptyset$	$\{A,B\}$	$\{A,B\}$	$\emptyset$
2	invoke	$\{A,B\}$	$\{A,B\}$	$\{f\}$	$\{A,B\}$	$\{A,B\}$	$\{f\}$
3	any	$\emptyset$	$\emptyset$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$
4	invoke	$\{A,B\}$	$\{B\}$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$
5	any	$\{A,B\}$	$\{B\}$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$
6	put(f)	$\{A\}$	$\{A\}$	$\emptyset$	$\{A,B\}$	$\{A,B\}$	$\{f\}$
7	any	$\{A,B\}$	$\{A,B\}$	$\{f\}$	$\{A,B\}$	$\{A,B\}$	$\{f\}$
8	return	$\{A,B\}$	$\{A,B\}$	$\{f\}$	$\{A,B\}$	$\{A,B\}$	$\{f\}$
9		$\{A,B\}$	$\{A,B\}$	$\{f\}$	$\{A,B\}$	$\{A,B\}$	$\{f\}$
10	return	$\{A,B\}$	$\{B\}$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$
11		$\{A,B\}$	$\{B\}$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$
12	return	$\{A,B\}$	$\{B\}$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$
13		$\{A,B\}$	$\{B\}$	$\emptyset$	$\{A,B\}$	$\{B\}$	$\emptyset$

(b) Least dataflow solution of the program example

Fig. 7. Program and analysis example

$instr(l)$ in a straightforward manner.

The transfer function F^{init} is defined with tree distinct cases. (i) In the first case, we are sure the class initializer C.<clinit> will be called because it has never been called before. We can hence use safely the last data flow of C.<clinit> but we combine it with a using the operator F_{call} described below. (ii) In the second case, we are sure that no class initializer will be called, either because there is no initialization edge at all, or there is one for a class C but we know that C.<clinit> has already been called. (iii) In the last case, the two previous cases may happen so we merge the corresponding data flows.

At last, F_{call} is an operator which combines data flows about calling contexts and calling returns. It allows to recover some *must* information that may have been discarded during the method call because of spurious calling contexts. It is based on the monotony of *Must* and *Wf*: these sets are under-approximations of initialization history and initialized fields but since such sets only increase during execution a correct under-approximation (*Must*, *Wf*) at a point l is still a correct approximation at every point reachable from l.

The program example presented is Fig. 7 is given with the least solution of its corresponding dataflow problem. In this example, A.<clinit> has two potential callers in 1 and 7 but we don't propagate the dataflow facts from 7 to 6 because we know that A.<clinit> has already been called at this point, thanks to *Must*. At point 2, the method m is called but we don't propagate in $A_{\text{out}}(2)$ the exact values found in $A_{\text{in}}(m.last)$ because we would lose the fact that $A \in Must$ before the call. That is why we combine $A_{\text{in}}(2)$ and $A_{\text{in}}(m.last)$ with F_{call} in order to refine the must information *Must* and *Wf*.

Theorem 4.4 (Computability) *The least data flow solution for the partial order $\sqsubseteq$ is computable by the standard fixpoint iteration techniques.*

Proof. This is consequence of the facts that each equation is monotone, there is a finite number of program points in p and $(A^\sharp, \sqsubseteq, \sqcup, \sqcap)$ is a finite lattice. □

Theorem 4.5 (Soundness) *If* $(A_{\text{in}}, A_{\text{out}})$ *is a data flow solution then for all reachable states* $<i, cs, s, h> \in [\![p]\!]$, $A_{\text{in}}(i) \sim (s, h)$ *holds.*

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

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 (`IndexOutOfBoundException`, `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 `NullPointerException`s and [1] to remove the `IndexOutOfBoundException`s 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 `<clinit>` method to the `<clinit>` method of its superclass (except for `Object.<clinit>`). 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 `<clinit>` method such that, if l is the label we introduce at the beginning of `C.<clinit>`, then $\mathit{flow}_{\text{clinit}}(l) = \mathit{super}(C)$.

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]).

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 *Wf* either when the corresponding class is added to *Must* or at the very beginning ($m_o.first$). 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 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 `C.f` is read at program point l, we need to check that `C.f` is in the set of written field at this particular program point.

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 Leino in [5], our work in [10] or Fähndrich and Xia in [6]. Other work have been focused on different properties such as Unkel 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 Debbabi *et al.* in [4] or Belblidia and Debbabi 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 Soffa [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 problems 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.

8 Conclusion and Future Work

We have shown that class initialization is a complex mechanism and that, although in most cases it works as excepted, 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

[1] Bodik, R., R. Gupta and V. Sarkar, *ABCD: eliminating array bounds checks on demand*, in: *Proc. of PLDI*, 2000, pp. 321–333.

[2] Börger, E. and W. Schulte, *A programmer friendly modular definition of the semantics of java*, in: *Formal Syntax and Semantics of Java*, LNCS (1999), pp. 353–404.

[3] Börger, E. and W. Schulte, *Initialization problems for java*, Software - Concepts and Tools **19** (2000), pp. 175–178.

[4] Debbabi, M., N. Tawbi and H.Yahyaoui, *A formal dynamic semantics of java: An essential ingredient of java security*, Journal of Telecommunications and Information Technology **4** (2002), pp. 81–119.

[5] Fähndrich, M. and K. R. M. Leino, *Declaring and checking non-null types in an object-oriented language*, ACM SIGPLAN Notices **38** (2003), pp. 302–312.

[6] Fähndrich, M. and S. Xia, *Establishing object invariants with delayed types*, in: *Proc. of OOPSLA* (2007), pp. 337–350.

[7] Harrold, M. J. and M. L. Soffa, *Efficient computation of interprocedural definition-use chains*, TOPLAS **16** (1994), pp. 175–204.

[8] Hirzel, M., A. Diwan and M. Hind, *Pointer analysis in the presence of dynamic class loading*, in: *Proc. of ECOOP*, 2004, pp. 96–122.

[9] Hubert, L., *A Non-Null annotation inferencer for Java bytecode*, in: *Proc. of PASTE* (2008), (To Appear).

[10] Hubert, L., T. Jensen and D. Pichardie, *Semantic foundations and inference of non-null annotations*, in: *Proc. of FMOODS*, LNCS **5051** (2008), pp. 132–149.

[11] Kozen, D. and M. Stillerman, *Eager class initialization for java*, in: *Proc. of FTRTFT* (2002), pp. 71–80.

[12] Lindholm, T. and F. Yellin, "The Java™ Virtual Machine Specification, Second Edition," Prentice Hall PTR, 1999.

[13] Livshits, B., J. Whaley and M. S. Lam, *Reflection analysis for java*, in: *Proc of APLAS*, 2005.

[14] N. Belblidia, M. D., *A dynamic operational semantics for JVML*, Journal of Object Technology **6** (2007), pp. 71–100.

[15] Necula, G., *Proof-Carrying Code*, in: *Proc. of POPL* (1997), pp. 106–119.

[16] Unkel, C. and M. S. Lam, *Automatic inference of stationary fields: a generalization of java's final fields*, in: *Proc. of POPL* (2008), pp. 183–195.

User-Definable Resource Usage Bounds Analysis for Java Bytecode

Jorge Navas[1,4]

[1] *School of Computing*
National University of Singapore
Republic of Singapore

Mario Méndez-Lojo[2,4]

[2] *Department of Computer Science*
University Texas at Austin
Austin, TX (USA)

Manuel V. Hermenegildo[3,4,5]

[3] *IMDEA-Software, Madrid (Spain),*
Departments of Computer Science
[4] *University of New Mexico, Albuquerque, NM (USA)*
and [5] *Technical 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, bytes sent or received by an application, number of files left open, number of SMSs sent or received, number of accesses to a database, money spent, energy consumption, etc. We present 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. 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 proposed is independent of the particular resource. We also present some experimental results from a prototype implementation of the approach covering a significant set of interesting resources.

1 Introduction

The usefulness of analyses which can infer information about the costs of computations is widely recognized since 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., [27,34,36,20,9,21,40] for functional languages, [38,8,19,42] for imperative languages, and [17,16,18,32] 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 (e.g., [14,6,12,5,22]). Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent rather than

[1] Email: navas@comp.nus.edu.sg
[2] Email: marioml@ices.utexas.edu
[3] Email: herme@fi.upm.es

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

(or, rather, in addition to) the predefined, more traditional costs such as steps, time, or memory. Regarding the object of certification, in the case of mobile code the certification and checking process is often performed at the bytecode level [28], since, in addition to other reasons of syntactic convenience, bytecode is what is most often available at the receiving (checker) end.

We propose a fully automated framework which infers upper bounds on the usage that a Java bytecode program makes of *application programmer-definable* resources. Examples of such programmer-definable resources are bytes sent or received by an application over a socket, number of files left open, number of SMSs sent or received, number of accesses to a database, number of licenses consumed, monetary units spent, energy consumed, disk space used, and of course, execution steps (or bytecode instructions), time, or memory. A key issue in approach is that resources are defined by programmers and by means of *annotations*. The annotations defining each resource must provide for some relevant user-selected elements corresponding to the bytecode program being analyzed (classes, methods, variables, etc.), a value that describes the cost of that element for that particular resource. These values can be constants or, more generally, functions of the input data sizes. The objective of our analysis is then to statically derive from these elementary costs an upper bound on the amount of those resources that the program as a whole (as well as individual blocks) will consume or provide.

Our approach builds on the work of [17,16] for logic programs, where cost functions are inferred by solving recurrence equations derived from the syntactic structure of the program. Most previous work deals only with concrete, traditional resources (e.g., execution steps, time, or memory). The analysis of [32] also allows program-level definition of resources, but it is designed for Prolog and works at the source code level, and thus is not directly applicable to Java bytecode due to particularities like virtual method invocation, unstructured control flow, assignment, the fact that statements are low-level bytecode instructions, the absence of backtracking (which has a significant impact on the method used in [32]), etc. Also, the presentation of [32] is descriptive in contrast to the concrete algorithm provided herein. In [2], a cost analysis is described that does deal with Java bytecode and is capable of deriving cost relations which are functions of input data sizes. The authors also presented in [3] an experimental evaluation of the approach. This approach is generic, in the same sense as, e.g., [16], in that both the conceptual framework and its implementation allow adaptation to different resources. However, this is done typically in the implementation. Our approach is interesting in that it allows the application programmer to define the resources through annotations directly in the Java source, and without changing the analyzer code or tables in any way. Also, without claiming it as any significant contribution of course, we provide for implementation convenience a somewhat more concrete, algorithmic presentation, in contrast to the more descriptive approach of previous work (including [17,16,32,2,3], etc.).

2 User-Defined Resources: Overview of the Approach

A *resource* is a fundamental component in our approach. A resource is a user-defined notion which associates a basic cost function with some user-selected elements (class,

```
import java.net.URLEncoder;
public class CellPhone {
 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
  if (smsPk != null) {
    String newSms = enc.format(smsPk.sms);
    stm.send(newSms);
    smsPk.next=sendSms(smsPk.next,enc,stm);
    smsPk.sms = newSms;
  }
  return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=size(s)")
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  @Cost({"cents","2*size(data)"})
  native void send(String data);
}
```

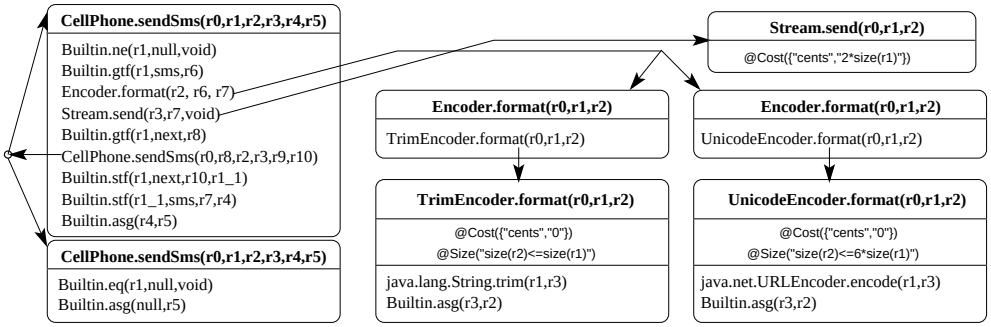


Fig. 1. Motivating example: Java source code and Control Flow Graph

method, statement) in the program. This is expressed by adding *Java annotations* to the code. The objective of the analysis is to approximate the usage that the program makes of the resource.

We start by illustrating the overall approach through a working example. The Java program in Fig. 1 emulates the process of sending text messages within a cell phone. This example 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 provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The phone (class `CellPhone`) receives a list of packets (`SmsPacket`), each one containing a single SMS, encodes them (`Encoder`), and sends them through a stream (`Stream`). There are two types of encoding: `TrimEncoder`, which eliminates any leading and trailing white spaces, and `UnicodeEncoder`, which converts any special character into its Unicode($\backslash uxxxx$) equivalent. The length of the SMS which the cell phone ultimately sends through the stream depends on the size of the encoded message.

In the example, the resource is the cost in cents of a dollar for sending the list of text messages. We will assume for the sake of discussion that the carrier charges are proportional to the number of characters sent, and at 2 cents/character. This is reflected by the user in the method that is ultimately responsible for the communication (`Stream.send`), by adding the annotation `@Cost({"cents","2*size(data)"})`. Similarly, the formatting of an SMS made in

any implementation of `Encoder.format` is free, as indicated by the `@Cost({"cents","0"})` annotation (the actual system allows defining overall cost defaults 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 resources to be tracked (which is done by simply mentioning them in the annotations) 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 resource). 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 $\mathcal{L}$):

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid (\sum \mid \prod) \langle expr \rangle \\
& \mid & \langle expr \rangle^{\langle expr \rangle} \mid log_{num} \langle expr \rangle \mid -\langle expr \rangle \\
& \mid & \langle expr \rangle ! \mid \infty \mid \text{num} \\
& \mid & \text{size}([\langle measure \rangle,]\text{arg(r num)}) \\
\langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\
\langle measure \rangle & ::= & \textbf{int} \mid \textbf{ref} \mid \ldots
\end{array}
$$

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 intermediate 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 s_{r_i} we denote the size of input formal parameter position i, corresponding to variable r_i):

$$\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) \leq 0$$
$$\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 7 \times s_{r_1} - 6 + \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})$$

The size of the returned value r_5 is independent of the sizes of the input parameters *this*, *enc*, and *stm* (s_{r_0}, s_{r_2} and s_{r_3} respectively) but not of the size s_{r_1} of the list of text messages *smsPk* (r_1 in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. In the example in Fig. 1, the user indicates that the formatting in `UnicodeEncoder` results in strings that are at most six times longer than the ones received as input `@Size("size(ret)<=6*size(s)")`, while the trimming in `TrimEncoder` returns strings that are equal or shorter than the input (`@Size("size(ret)<=size(s)")`). In this case the equations provide implicitly the size measure (i.e., that the size of a string is its length). The equation system shown above is approximated by a recurrence solver included in our analysis in order to obtain the closed form solution $\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 3.5 \times s^2_{r_1} - 2.5 \times s_{r_1}$. This is a reasonable bound given that we have not specified a maximum size for each string.

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

$$Cost_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) \leq 0$$
$$Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 12 \times s_{r_1} - 12 + Cost_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})$$

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 $Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 6 \times s^2_{r_1} - 6 \times s_{r_1}$.

3 Intermediate program representation

Analysis of a Java bytecode program normally requires its translation into an intermediate 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 decompilation 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-

tion has some limitations: it does not yet support reflection, threads, or runtime exceptions. The following grammar describes the intermediate representation; some of the elements in the tuples are named so we can refer to them as *node*.name.

$$
\begin{array}{lcl}
CFG & ::= & Block^{+} \\
Block & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{fpars}{:}Id^{+},\mathsf{annot}{:}expr^{*},\mathsf{body}{:}Stmt^{*}) \\
Sig & ::= & (\mathsf{class}{:}Type,\mathsf{name}{:}Id,\mathsf{pars}{:}Type^{+}) \\
Stmt & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{apars}{:}(Id|Ct)^{+}) \\
Var & ::= & (\mathsf{name}{:}Id,\mathsf{type}{:}Type)
\end{array}
$$

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, unless 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 `asg`, field dereference `gtf`, 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* $stmt_1$ `else` $stmt_2$ in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of decompiling *cond*, while we attach $\overline{cond}$ 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 preserved in the bytecode. Annotations are carried over to our CFG representation, as can be seen in Fig. 1.

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 r_0, r_1, r_2, r_3 correspond to $this, smsPk, enc$, and stm, respectively. In the case of r_1, 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) builtin block methods. The output formal parameter r_4 contains the final state of r_1 after those modifications. The value returned by the block methods is contained in r_5. 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`}.

```
resourceAnalysis(CFG, res) {
 CFG←classAnalysis(CFG)
 Aliases←aliasAnalysis(CFG)
 mt←initialize(CFG)
 dg←dataDependencyAnalysis(CFG, Aliases ,mt)
 for (SCC:SCCs)
   //in reverse topological order
   mt←genSizeEqs(SCC,mt,CFG,dg)
   mt←genResourceUsageEqs(SCC,res ,mt,CFG)
 return mt
}
```

```
normalize(Eqs) {
 for (size relation p ≤ e1 : Eqs)
   do
     if (expression s appears in e1
         and s ≤ e2 ∈Eqs)
       replace ocurrences of s in e1 with e2
   while there is change
 return Eqs
}
```

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 dependencies 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 graph*s for the block methods within a strongly connected component. Each graph $G = (V, E)$ represents data dependencies between positions corresponding to statements in the same block method, including its formal parameters. Vertexes in V denote positions, and edges $(s_1, s_2) \in E$ denote that s_2 is dependent on s_1 (s_1 is a *predecessor* of s_2). We will assume a `predec` function that takes 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 `TrimEncoder.format` block method.

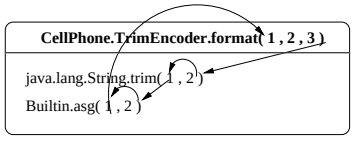


Fig. 3:

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

to represent input and output size relationships for each statement as a function defined in terms of the formal parameter sizes. Unless otherwise stated, whenever we refer to a parameter we mean its position.

The size of an input is defined in terms of measures. By *measure* we mean a function that, given a data structure, returns a number. Our method is parametric on measures, which can be defined by the user and attached via annotations to parameters or classes. For concreteness, we have defined herein two measures, int for integer variables, and the *longest path-length* [37,2] ref for reference variables. The longest path-length of a variable is the cardinality of the longest chain of pointers than can be followed from it. More complex measures can be defined to handle other data types such as cyclic structures, arrays, etc. The set of measures will be denoted by $\mathcal{M}$.

The size analysis algorithm is given in pseudo-code in Fig. 4; its main steps are:

(i) Assign an upper bound to the size of every parameter position of all statements, including formal parameters, for all the block methods with the same signature (genSigSize).

(ii) For a given signature, take the set of size inequations returned by (i) and rename each size relation in terms of the sizes of input formal parameters (normalize).

(iii) Repeat the first step for every signature in the same strongly-connected component (genSizeEqs).

(iv) Simplify size relationships by resolving mutually recursive functions, and find closed form solutions for the output formal parameters (genSizeEqs).

Intermediate results are cached in a memo table mt, which for every parameter position stores measures, sizes, and resource usage expressions defined in the $\mathcal{L}$ language.

The size of the parameter at position i in statement $stmt$, under measure m, is referred to as $\texttt{size}(m, stmt, i)$. We consider a parameter position to be *input* if it is bound to some data when the statement is invoked. Otherwise, it is considered an *output parameter position*. In the case of input parameter and output formal parameter positions, an upper bound on that size is returned by getSize (Fig. 4). The upper bound can be a concrete value when there is a constant in the referred position, i.e., when the `val` function returns a non-infinite value:

Definition 4.1 The concrete size value for a parameter position under a particular measure is returned by $\texttt{val} : \mathcal{M} \times \mathcal{S}tmt \times \mathbb{N} \rightarrow \mathcal{L}$, which evaluates the *syntactic* content of the actual parameter in that position:

$$\texttt{val}(m, stmt, i) = \begin{cases} n & \textit{if } stmt.\mathit{apars}_i \textit{ is an integer } n \textit{ and } m{=}\textbf{\textit{int}} \\ 0 & \textit{if } stmt.\mathit{apars}_i \textit{ is } \texttt{null} \textit{ and } m{=}\textbf{\textit{ref}} \\ \infty & \textit{otherwise} \end{cases}$$

If the content of that input parameter position is a variable, the algorithm searches the data dependency graph for its immediate predecessor. Since the intermediate representation is in SSA form, the only possible scenarios are that either

```
genSizeEqs(SCC, mt, CFG, dg) {
  Eqs ← ∅^|SCC|
  for (sig: SCC)
    Eqs[sig] ← genSigSize(sig, mt, SCC, CFG, dg)
  Sols ← recEqsSolver(simplifyEqs(Eqs))
  for (sig:SCC)
    insert(mt, size, sig, Sols[sig])
  return mt
}

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

genBlockSize(bm, mt, SCC, dg) {
  Eqs ← ∅
  for (stmt:bm.body)
    I ← stmt input parameter positions
    Eqs ← Eqs ∪ genInSize(stmt, I, mt, dg)
    Eqs ← Eqs ∪ genOutSize(stmt, mt, SCC)
  K ← bm output formal parameter positions
  Eqs ← Eqs ∪ genInSize(bm, K, mt, dg)
  return Eqs
}

genInSize(elem, Pos, mt, dg) {
  Eqs ← ∅
  for (pos:Pos)
    m ← lookup(mt, measure, elem.sig, pos)
    s ← getSize(m, elem.id, pos, dg)
    Eqs ← Eqs ∪ {size(m, elem.id, pos) ≤ s}
  return Eqs
}
```

```
genOutSize(stmt, mt, SCC) {
  {i_1, ..., i_l} ← stmt input positions
  sig ← stmt.sig
  {m_{i_1}, ..., m_{i_l}} ← {lookup(mt, measure, sig, i_1), ...,
                             lookup(mt, measure, sig, i_l)}
  {s_{i_1}, ..., s_{i_l}} ← {size(m_{i_1}, stmt.id, i_1), ...,
                             size(m_{i_l}, stmt.id, i_l)
  Eqs ← ∅
  O ← stmt output parameter positions
  for (o:O)
    m_o ← lookup(mt, measure, sig, o)
    if (sig ∉ SCC)
      Size_user ← A^o_sig(s_{i_1}, ..., s_{i_l})
      Size_alg' ← max(lookup(mt, size, sig, o))
      Size_alg ← Size_alg'(s_{i_1}, ..., s_{i_l})
      Size_o ← min(Size_user, Size_alg)
    else
      Size_o ← Size^o_sig(m_o, s_{i_1}, ..., s_{i_l})
    Eqs ← Eqs ∪ {size(m_o, stmt.id, o) ≤ Size_o}
  return Eqs
}

getSize(m, id, pos, dg) {
  result ← val(m, id, i)
  if (result ≠ ∞)
    return result
  else
    if (∃(elem, pos_p) ∈ predec(dg, id, pos))
      m_p ← lookup(mt, measure, elem.sig, pos_p)
      if (m = m_p)
        return size(m_p, elem.id, pos_p)
  return ∞
}
```

Fig. 4. The size analysis algorithm

there is a unique predecessor whose size is assigned to that input parameter position, or there is none, causing the input parameter size to be unbounded (∞).

Consider now an output parameter position within a block method, case covered in genOutSize (Fig. 4). If the output parameter position corresponds to a non-recursive invoke statement, either a size relationship function has already been computed recursively (since the analysis traverses each strongly-connected component in reverse topological order), or it is provided by the user through size annotations. In the first case, the size function of the output parameter position can be retrieved from the memo table by using the lookup operation, taking the maximum in case of several size relationship functions, and then passing the input parameter size relationships to this function to evaluate it. In the second scenario, the size function of the output parameter position is provided by the user through size annotations, denoted by the $\mathcal{A}$ function in the algorithm. In both cases, it will able to return an explicit size relation function.

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);
```

```
    // ...rest of annotated builtins
}
```

which results in equation $\mathcal{A}^1_{\mathtt{asg}}(\mathtt{ref}, \mathtt{size}(\mathtt{ref}, asg, 0)) \leq \mathtt{size}(\mathtt{ref}, asg, 0)$.

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, those 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 simplifyEqs procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver (recEqsSolver, called from genSizeEqs) 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 algorithm with an example of how the size relations are inferred for the two `CellPhone.sendSms` block methods (Fig. 1), using the ref measure for reference variables. We will refer to the k-th occurrence of a statement *stmt* in a block method as $stmt_k$, and denote `CellPhone.sendSms`, `Encoder.format`, and `Stream.send` by `sendSms`, `format`, and `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_{r_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_{r_i}). Also, in the first block of rows we observe that the algorithm has returned $6 \times \mathtt{size}(\mathtt{ref}, format, 1)$ as upper bound for the size of the formatted string, $\mathsf{max}(\mathsf{lookup}(mt, \mathtt{size}, format, 2))$. The result is

Size parameter relationship equations (normalized)

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, ne, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, ne, 1) & \leq \texttt{val}(\texttt{ref}, ne, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, gtf_1, 0) & \leq \texttt{size}(\texttt{ref}, ne, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_1, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_1, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 1) & \leq \texttt{size}(\texttt{ref}, gtf_1, 2) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 2) & \leq \texttt{max}(\texttt{lookup}(mt, \texttt{size}, format, 2))(\texttt{size}(\texttt{ref}, format, 2)) \\
& \leq \texttt{max}(s_{r1}, 6 \times s_{r1})(s_{r_1} - 1) \\
& \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, send, 1) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, gtf_2, 0) & \leq \texttt{size}(\texttt{ref}, gtf_1, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_2, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_2, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, sendSms, 5) & \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, \_, \texttt{size}(\texttt{ref}, sendSms, 1), \_, \_) \\
& \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 0) & \leq \texttt{size}(\texttt{ref}, gtf_2, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, stf_1, 2) & \leq \texttt{size}(\texttt{ref}, sendSms, 5) \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_1, 0), \_, \texttt{size}(\texttt{ref}, stf_1, 2)) \\
& \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 0) & \leq \texttt{size}(\texttt{ref}, stf_1, 3) \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 2) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, stf_2, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_2, 0), \_, \texttt{size}(\texttt{ref}, stf_2, 2)) \\
& \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{size}(\texttt{ref}, stf_2, 3) \\
& \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \\
& \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})
\end{array}
$$

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, eq, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, eq, 1) & \leq \texttt{val}(\texttt{ref}, eq, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{val}(\texttt{ref}, asg, 0) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq 0
\end{array}
$$

Output parameter size functions for builtins (provided through annotations)

$$
\begin{array}{rl}
\mathcal{A}^2_{\texttt{gtf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf, 0), \_) & \leq \texttt{size}(\texttt{ref}, gtf, 0) - 1 \\
\mathcal{A}^1_{\texttt{asg}}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) & \leq \texttt{size}(\texttt{ref}, asg, 0) \\
\mathcal{A}^3_{\texttt{stf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf, 0), \_, \texttt{size}(\texttt{ref}, stf, 2)) & \leq \texttt{size}(\texttt{ref}, stf, 0) + \texttt{size}(\texttt{ref}, stf, 2)
\end{array}
$$

Simplified size equations and closed form solution

$$
\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}
$$

$$
\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 3.5 \times s^2_{r1} - 2.5 \times s_{r1}
$$

Fig. 5. Size equations example

the maximum of the two upper bounds given by the user for the two implementations for `Encoder.format` since `TrimEncoder.format` eliminates any leading and trailing white spaces (thus the output is at most as bigger as the input), whereas `UnicodeEncoder.format` converts any special character into its Unicode equivalent (thus the output is at most six times the size of the input), a safe upper bound for the output parameter position size is given by the second annotation.

In the particular case of builtins and methods for which we do not have the code, size relationships are not computed but rather taken from the user `@Size` annotations. These functions are illustrated in the third block of rows. Finally, in the fourth block of rows we show the recurrence equations built for the output

```
genResourceUsageEqs(SCC, res, mt, CFG) {
  Eqs ← ∅^|SCC|
  for (sig : SCC)
    Eqs[sig] ← genSigRU(sig, res, mt, SCC, CFG)
  Sols ← recEqsSolver(simplifyEqs(Eqs))
  for (sig : SCC)
    insert(mt, cost, max(Sols[sig]))
  return mt
}

genSigRU(sig, res, mt, SCC, CFG) {
  Eqs ← ∅
  BMs ← getBlocks(CFG, sig)
  for (bm : BMs)
    body ← bm.body
    Cost_body ← 0
    for (stmt : body)
      Cost_stmt ← genStmtRU(stmt, res, mt, SCC)
      Cost_body ← Cost_body + Cost_stmt
    Cost_bm ← genBlockRU(bm, res, mt)
    Eqs ← Eqs ∪ {Cost_bm ≤ Cost_body}
}

genStmtRU(stmt, res, mt, SCC) {
  {i_1, ..., i_k} ← stmt input parameter positions
  {s_i1, ..., s_ik} ←
      {max(lookup(mt, size, stmt.sig, i_1)), ...,
       max(lookup(mt, size, stmt.sig, i_k))}
  if (stmt.sig ∉ SCC)
    Cost_user ← A_stmt.sig(res, s_i1, ..., s_ik)
    Cost_alg' ← lookup(mt, cost, res, stmt.sig)
    Cost_alg ← Cost_alg'(s_i1, ..., s_ik)
    return min(Cost_alg, Cost_user)
  else return Cost(stmt.sig, res, s_i1, ..., s_ik)
}

genBlockRU(bm, res, mt) {
  {i_1, ..., i_l} ← bm input formal parameter positions
  {s_i1, ..., s_il} ←
      {lookup(mt, size, bm.id, i_1), ...,
       lookup(mt, size, bm.id, i_l)
  return Cost(bm.id, res, s_i1, ..., s_il)
}
```

Fig. 6. The resource usage analysis algorithm

parameter sizes in the block method and in the final row the closed form solution obtained.

4.2 Resource usage analysis

The core of our framework is the resource usage analysis, whose pseudo code is shown in Fig 6. It takes a strongly-connected component of the CFG, including the set of annotations which provide the application programmer-provided resources and cost functions, and calculates a resource usage function which is an upper bound on the usage made by the program of those resources. The algorithm manipulates the same memo table described in Sec. 4.1 in order to avoid recomputations and access the size relationships already inferred.

The algorithm is structured in a very similar way to the size analysis (which also allows us to draw from it to keep the explanation within space limits): for each element of the strongly-connected component the algorithm will construct an equation for each block method that shares the same signature representing the resource usage of that block. To do this, the algorithm will visit each invoke statement. There are three possible scenarios, covered by the genStmtRU function. If the signatures of caller and callee(s) belong to the same strongly-connected component, we are analyzing a recursive invoke statement. Then, we add to the body resource usage a symbolic resource usage function, in an analogous fashion to the case of output parameters in recursive invocations during the size analysis.

The other scenarios occur when the invoke statement is non-recursive. Either a resource usage function $Cost_{alg}$ for the callee has been previously computed, or there is a user annotation $Cost_{usr}$ that matches the given signature, or both. In the latter case, the minimum between these two functions is chosen (i.e., the most precise safe upper bound assigned by the analysis to the resource usage of the non-recursive invoke statement) or a warning is issued.

Example 4 The call (sixth statement) in the upper-most `CellPhone.sendSms`

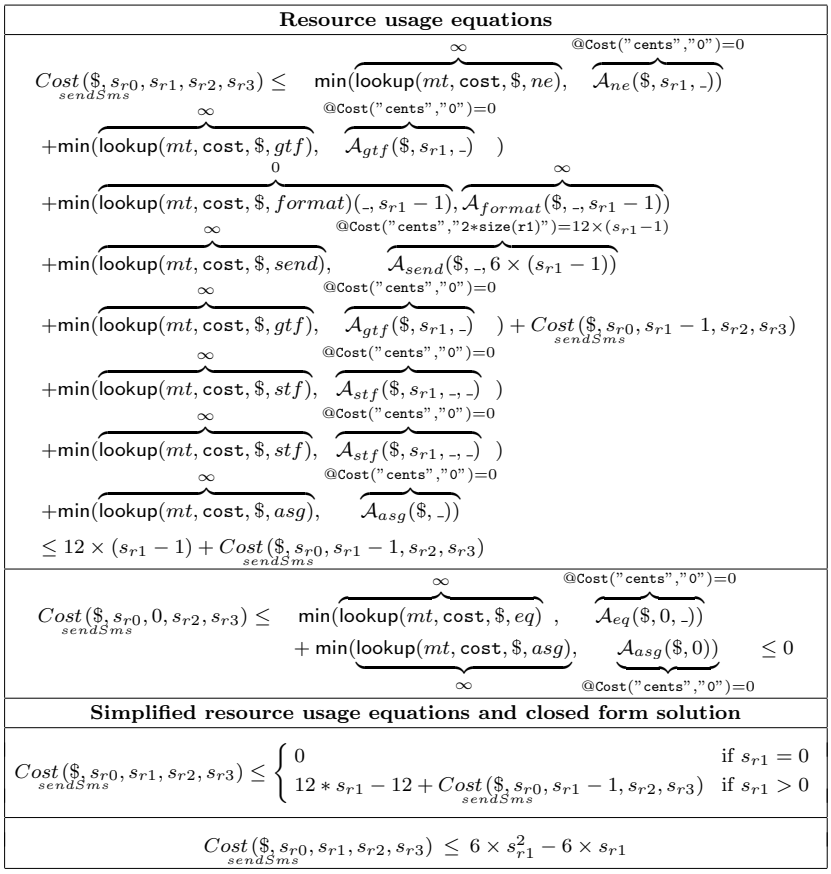


Fig. 7. Resource equations example

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 genStmtRU is $Cost_{sendSms}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$. Note that the input size relationships were already normalized during the size analysis. Now consider the invocation of `Stream.send`. The resource usage expression for the statement is defined by the function $\mathcal{A}_{send}(\$, \_, 6 \times (s_{r1}-1))$ 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({"cents","2*size(r1)"})` attached to the block method describing the behavior of $\mathcal{A}_{send}$ and yielding the expression $Cost_{user} = 12 \times (s_{r1}-1)$. On the other hand, the absence of any callee code to analyze –the original method is native– results in $Cost_{alg} = \infty$. Then, the upper bound obtained by the analysis for the statement is $\min(Cost_{alg}, Cost_{user}) = Cost_{user}$.

At this point, the analysis has built a resource usage function (denoted by $Cost_{body}$) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form $Cost_{block} \leq Cost_{body}$ where $Cost_{block}$ 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 simplifyEqs and, finally, they are solved calling recEqsSolver, both already defined in Sec. 4.1. This process yields an (in gen-

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., $Cost_{sendSms}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3})$ and $Cost_{sendSms}(\$, s_{r0}, 0, s_{r2}, s_{r3})$), 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 $6 \times s_{r1}^2 - 6 \times s_{r1}$.

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_s 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_r 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 have we represented 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

Program	Resource(s)	t_s	t_r	t	Resource Usage Func.	
BST	Heap usage	250	22	367	$O(2^n)$	$n \equiv$ tree depth
CellPhone	SMS monetary cost	271	17	386	$O(n^2)$	$n \equiv$ packets length
Client	Bytes received and	391	38	527	$O(n)$	$n \equiv$ stream length
	bandwidth required				$O(1)$	—
Dhrystone	Energy consumption	602	47	759	$O(n)$	$n \equiv$ int value
Divbytwo	Stack usage	142	13	219	$O(log_2(n))$	$n \equiv$ int value
Files	Files left open and	508	53	649	$O(n)$	$n \equiv$ number of files
	Data stored				$O(n \times m)$	$m \equiv$ stream length
Join	DB accesses	334	19	460	$O(n \times m)$	$n, m \equiv$ records in tables
Screen	Screen width	388	38	536	$O(n)$	$n \equiv$ stream length

Table 1
Times of different phases of the resource analysis and resource usage functions.

code) we kept track of the number of file descriptors left open. The data inferred for this resource can be clearly useful, e.g., for debugging: the resource usage function inferred in this case ($O(n)$) denotes that the programmer did not close $O(n)$ file descriptors previously opened. In program *Join* (a database transaction which carries out accesses to different tables) we decided to measure the number of accesses to such external tables. This information can be used, e.g., for resource-oriented specialization in order to perform optimized checkpoints in transactional systems. The rest of the benchmarks include other definitions of resources which are also typically useful for verifying application-specific properties: *BST* (a generic binary search tree, used in [4] where a heap space analysis for Java bytecode is presented), *CellPhone* (extended version of program in Figure 1), *Client* (a socket-based client application), *Dhrystone* (a modified version of a program from [25] where a general framework is defined for estimating the energy consumption of embedded JVM applications; the complete table with the energy consumption costs that we used can be found there), *DivByTwo* (a simple arithmetic operation), and *Screen* (a MIDP application for a cellphone, where the analysis is used to make sure that message lines do not exceed the phone screen width). The benchmarks also cover a good range of complexity functions ($O(1), O(log(n), O(n), O(n^2) \ldots, O(2^n), \ldots$) and different types of structural recursion such as simple, indirect, and mutual.

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

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

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*, LNCS 5079, pages 221–237, 2008.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 67–83. Elsevier - North Holland, July 2007.

[4] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[5] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[6] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.

[7] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[8] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing*, Apr. 2002.

[9] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[11] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC*, pages 234–250, 1994.

[12] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.

[13] S.J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[14] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[15] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*, pages 77–101, 1995.

[16] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.

[17] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.

[18] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.

[19] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proc. of DDECS*. IEEE Computer Society, 2006.

[20] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.

[21] B. Grobauer. Cost recurrences for DML programs. In *Int'l. Conf. on Functional Programming*, pages 253–264, 2001.

[22] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.

[23] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.

[24] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[25] Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

[26] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, 1992.

[27] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *TOPLAS*, 10(2), 1988.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[29] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.

[30] M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[31] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

[32] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, LNCS, 2007.

[33] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of PPDP'06*, pages 261–271. ACM Press, 2006.

[34] M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[35] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented*, pages 43–55, 2001.

[36] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[37] F. Spoto, P.M. Hill, and E. Payet. Path-length analysis of object-oriented programs. In *EAAI'06*, ENTCS. Elsevier, 2006.

[38] Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour*, 2004.

[39] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[40] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

[41] H. S. Wilf. *Algorithms and Complexity*. A.K. Peters Ltd, 2002.

[42] R. Wilhelm. Timing analysis and timing predictability. In *Proc. FMCO*, LNCS. Springer-Verlag, 2004.

An Ahead-of-time Yet Context-Sensitive Points-to Analysis for Java

Xin Li[1,2]

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

Mizuhito Ogawa[3]

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 all-in-one weighted pushdown model checking. The analysis is context-sensitive in the sense that, (i) method calls and returns match with each other (a.k.a., *valid 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 is that, by encoding dataflow as weights, invalid control flows that violate Java semantics on dynamic dispatch are detected as those carrying conflicted dataflow. Our analysis is presented as field-sensitive and flow-sensitive. Flow-insensitivity is shown to be easily obtained as a hierarchy considering efficiency and concurrent behaviors. 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: Points-to Analysis, Weighted Pushdown Model Checking, Java

1 Introduction

Points-to analysis [3] 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 Evolvable 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

depend on points-to analysis to reason the underlying control/data flow of Java programs. Due to dynamic object-oriented features like *dynamic dispatch* [4], 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 providing 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 o^l an abstract heap object that is allocated at the program line l, and by $\mapsto$ the mapping relation afterwards. An analysis will precisely compute $\{c \mapsto o^3, d \mapsto o^5\}$ if it obeys to valid paths, and will furthermore erroneously infer $\{c \mapsto o^5, d \mapsto o^3\}$ otherwise. An analysis will precisely compute $\{o^3.f \mapsto o^{15}, o^5.f \mapsto o^{20}\}$ if a context-sensitive call graph is constructed, and will furthermore erroneously infer $\{o^3.f \mapsto o^{20}, o^5.f \mapsto o^{15}\}$ otherwise.

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

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

[4] In this paper, we limit our focus to single dynamic dispatch only. Multiple dynamic dispatch, e.g., reflection in Java, demands non-trivial extension and thus independent discussion.

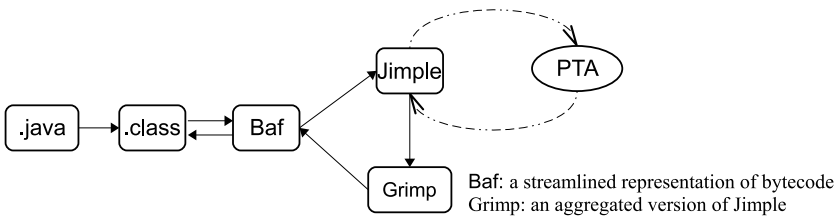


Fig. 1. Points-to Analysis on Jimple in the Soot Framework

been originally designed to simplify the process of developing new optimizations for Java bytecode and supports three kinds of intermediate representations of bytecode. As shown in Figure 1, the Java source code (bytecode) is firstly compiled into the Jimple [25] code, which is a typed three-address intermediate representation. Our analysis PTA is then performed on the Jimple code, and the analysis results can be encoded into the class file as attributes for any kind of later use. It can be useful for other occasions to perform complicated static analysis on an intermediate language like Jimple and annotate the class file with analysis results.

The remainder of the paper is organized as follows: Section 2 briefly introduces weighted pushdown model checking. Section 3 formalizes our abstraction and modelling on heap operations. Section 4 presents our ahead-of-time points-to analysis as all-in-one weighted pushdown model checking. The skeleton of holding soundness property is given, and the prototype implementation is shown. Section 5 compares related work and Section 6 concludes this paper with a discussion on future work.

2 Background

2.1 Weighted Pushdown Model Checking

Definition 2.1 A **pushdown system** $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown automaton regardless of input, where Q is a finite set of states called control locations, and Γ 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 $w_0 \in \Gamma^*$ are the initial control location and stack contents respectively. We denote the transition rule $((q_1, w_1), (q_2, w_2)) \in \Delta$ by $\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_2 \rangle$. A **configuration** of P is a pair $\langle q, w \rangle$, where $q \in Q$ and $w \in \Gamma^*$. Δ defines the transition relation $\Rightarrow$ between pushdown configurations such that if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \Rightarrow \langle q, \omega\omega' \rangle$, 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** $S = (D, \oplus, \otimes, 0, 1)$ consists of a set D ($0, 1 \in D$) and two binary operations $\oplus$ and $\otimes$ 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) $(D, \otimes)$ is a monoid with 1 as the unit element;

(iii) $\otimes$ distributes over $\oplus$, i.e., $\forall a, b, c \in D$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$;

(iv) $\forall a \in D, a \otimes 0 = 0 \otimes a = 0$;

(v) The partial ordering $\sqsubseteq$ is defined on D such that $\forall a, b \in D, a \sqsubseteq b$ iff $a \oplus b = a$, and there are no infinite descending chains on D wrt $\sqsubseteq$.

Remark 2.3 As stated in Section 4.4 in [8], the distributivity of $\oplus$ can be loosened to $a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$. 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.

Definition 2.4 A **weighted pushdown system** is a triple $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f : \Delta \to D$ is a function that assigns a value from D to each rule of P.

Definition 2.5 Consider a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, and $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring. Assume $\sigma = [r_0, ..., r_k]$ to be a sequence of pushdown transition rules, where $r_i \in \Delta (0 \leq i \leq k)$, and $v(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$. Let path($c$,$c'$) be the set of all rule sequences that transform configurations from c into c'. The **generalized pushdown reachability problem**(GPR) is to find

$$\delta(c, C) = \bigoplus \{v(\sigma) | \sigma \in path(c, c'), c' \in C\}$$

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

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;
- $f \oplus g$ represents the merging of data flow at the meet of two control flows;
- $f \otimes g$ 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.

[5] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/
[6] http://www.cs.wisc.edu/wpis/wpds++/

Table 1
Syntactical Notations for References

f	$\in$	$\mathcal{F}$	Name constants of instance fields
v	$\in$	RefVar	Local variables and static fields
$v[i]$	$\in$	RefArr	Array references ($i \in \mathbb{Z}_0^+$)
v, $v[i]$, $v.f$	$\in$	$\mathcal{V}_{ref}$	RefVar $\cup$ RefArr $\cup$ RefVar $\times$ $\mathcal{F}$
v, $o[i]$, $o.f$	$\in$	$\mathcal{V}_{diref}$	RefVar $\cup$ $\mathcal{O} \times \mathbb{Z}_0^+ \cup \mathcal{O} \times \mathcal{F}$

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 automata. 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 $G = (N, E, n_0)$, where $N = N_i \cup N_c \cup N_e$ is the set of nodes, with N_i, N_c, N_e as the sets of internal nodes, call sites, and method exits, respectively. $E = E_i \cup E_c \cup E_e$ is the set of edges with $E_i \subseteq N_i \times N, E_c \subseteq N_c \times N_i, E_e \subseteq N_e \times N_i$, where E_i, E_c, and E_e are the sets of internal edges, call edges, and return edges, respectively. $n_0 \in N$ is the unique entry node of G. We denote by $N_r \subseteq N_i$ the set of return points of method calls. Let $\texttt{assign} : N_c \to N_r$ be the function that associates with each call site from N_c with a distinguished return point in N_r, $N_r = \{n_r \mid n_r = \texttt{assign}(n_c), n_c \in N_c\}$.

Definition 2.7 The encoding of an interprocedural control flow graph $G = (N, E, n_0)$ as a pushdown system $P = (Q, \Gamma, \Delta, q_0, w_0)$ is defined as follows

- Q is a singleton set denoted by $\{\cdot\}$;
- $\Gamma = N$ with $w_0 = n_0$;
- Δ is constructed as follows,

$$\begin{aligned}
&\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, n_i' \rangle && \text{if } (n_i, n_i') \in E_i \\
&\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, n_c n_r \rangle && \text{if } (n_i, n_c) \in E_c, \text{ and } n_r = \texttt{assign}(n_c) \in N_r \\
&\langle \cdot, n_e \rangle \hookrightarrow \langle \cdot, \epsilon \rangle && \text{if } (n_e, n_i) \in E_e
\end{aligned}$$

3 Modelling and Abstraction

3.1 Semantics of Heap Operations

Definition 3.1 Define $\mathcal{O}$ be the set of **heap objects** in the concrete domain, where $\top_o \in \mathcal{O}$ is the greatest element and represents any objects; $\bot_o \in \mathcal{O}$ is the least element and represents no objects (i.e., *null* reference). Elements in $\mathcal{O}$ except $\top_o$ and $\bot_o$ 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 $\mathcal{V}_{ref}$ that is syntactically allowed in Jimple; and (ii) the set of references $\mathcal{V}_{diref}$ in the semantic domain of *heap environments* (Definition 3.2). Static fields are treated in the same way with

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** $\texttt{henv}$ is a mapping from $\mathcal{V}_{diref}$ to $\mathcal{O}$. The set of heap environments is denoted by Henv^{con}. The evaluation function $\texttt{eval}^{con} : \text{Henv}^{con} \to \mathcal{V}_{ref} \to \mathcal{O}$ on reference variables is defined as:

$$\texttt{eval}^{con}(\texttt{henv}, v) = \texttt{henv}(v)$$

$$\texttt{eval}^{con}(\texttt{henv}, v[i]) = \texttt{henv}(\texttt{henv}(v)[i])$$

$$\texttt{eval}^{con}(\texttt{henv}, v.f) = \texttt{henv}(\texttt{henv}(v).f)$$

Let $\texttt{h}_{init}$ be the initial heap environment such that, for each $r \in \mathcal{V}_{diref}$, $\texttt{eval}^{con}(\texttt{h}_{init}, r) = \bot_o$ (*null* reference).

Let $\texttt{Loc}$ 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 $\texttt{next}(l)$ for $l \in \texttt{Loc}$. Later it will be discussed how to leverage the analysis to a flow-insensitive counterpart regarding concurrency.

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

- $States \subseteq (\texttt{Loc} \times \text{Henv}^{con})$ is a set of pairs of a program location and a heap environment,
- $\texttt{s}_{init}$ is the initial state, which is a pair of the program entry l_0 and $\texttt{h}_{init}$;
- $\to \ \subseteq States \times States$ is the set of operational semantic rules, and $\to^*$ denotes the transitive closure of $\to$.

A transition rule $\langle l, \texttt{henv}\rangle \to \langle \texttt{next}(l), \tau(\texttt{henv})\rangle$ for typical pointer assignment statements at $l \in \texttt{Loc}$ is shown in Table 2, where

- the function $\nu(\texttt{henv}, \text{T})$ generates a fresh heap object of type T in $\mathcal{O}$; and
- for $r,\ r' \in \mathcal{V}_{diref}$, $o \in \mathcal{O}$,

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

Definition 3.4 The **composition of heap environment transformers** is defined by the standard η-expansion, such that, for $\text{exph}_1, \text{exph}_2 \in \text{ExpHenv}$,

$$(\lambda\texttt{henv}.\ \text{exph}_2) \circ (\lambda\texttt{henv}.\ \text{exph}_1) =_\eta \lambda\texttt{h}.\ (\lambda\texttt{henv}.\ \text{exph}_2)(\lambda\texttt{henv}.\ \text{exph}_1)\texttt{h}$$

$$=_\beta \lambda\texttt{h}.\ \text{exph}_2[\texttt{henv} := \text{exph}_1[\texttt{henv} := \texttt{h}]]$$

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

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

Table 2
Heap Environment Transformer

Statement	Heap Environment Transformer
$x = \text{new T}$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \nu(\texttt{henv}, \text{T})]$
$x = y$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(y)]$
$x = y[n]$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{henv}(y)[n])]$
$x[n] = y$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [\texttt{henv}(x)[n] \mapsto \texttt{henv}(y)]$
$x = y.f$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{henv}(y).f)]$
$y.f = x$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [\texttt{henv}(y).f \mapsto \texttt{henv}(x)]$
$x = \text{return } y$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(y)]$
$x.m(r_0, ..., r_l)$	$\tau = \tau_0 \circ \tau_1 \circ ... \circ \tau_k$
	where $\tau_1, ..., \tau_k \in \texttt{Fun}$,
	$\texttt{Fun} = \{\lambda\texttt{henv}.\texttt{henv} \odot [\texttt{arg}_i \mapsto \texttt{henv}(m_i)] \mid$
	$r_i (0 \leq i \leq l)$ are arguments of reference type $\}$
	$\tau_0 = \lambda\texttt{henv}.\texttt{henv} \odot [\texttt{this} \mapsto \texttt{henv}(x)]$
	The method m to be invoked is from the class with which the
	type of $\texttt{henv}(x)$ 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 $\mathcal{T}$ and vice versa; `none conflicts` with t, for each t in $\mathcal{T}$ 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 reference, 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 $[\![\_]\!]$ refer to multi-array access. We denote $\{[\![o]\!] \mid o \in \texttt{Obj}\}$ by $[\![\texttt{Obj}]\!]$.

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** $\texttt{Obj} = \{\lceil t, l \rceil \mid t \in \mathcal{T}, l \in \texttt{Loc}\} \cup \{\top_{\texttt{obj}}, \bot_{\texttt{obj}}\}$, where $\top_{\texttt{obj}}$ is the greatest element and $\bot_{\texttt{obj}}$ is the least element. Other elements in $\texttt{Obj}$ except $\top_{\texttt{obj}}$ and $\bot_{\texttt{obj}}$ are incomparable.

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

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

$$\texttt{eval}^{abs}(\texttt{henv}, [\![v]\!]) = \{\texttt{henv}([\![o]\!]) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, v)\}$$

$$\texttt{eval}^{abs}(\texttt{henv}, v.f) = \{\texttt{henv}(o.f) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, v)\}$$

Table 3
Abstract Heap Environment Transformer ExpFun

Statement	Abstract Heap Environment Transformer
$x = \text{new T}$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \lceil t, l \rceil]$
$x = y$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(y)]$
$x = y[n]$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}([\![\texttt{henv}(y)]\!])]$
$x[n] = y$	$\lambda\texttt{henv}.\texttt{henv} \bullet [[\![\texttt{henv}(x)]\!] \mapsto \texttt{henv}(y)]$
$x = y.f$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(\texttt{henv}(y).f)]$
$y.f = x$	$\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{henv}(y).f \mapsto \texttt{henv}(x)]$

Let $\texttt{h}_{init}$ be the abstract initial heap environment such that $\texttt{eval}^{abs}(\texttt{h}_{init}, r) = \bot_{\texttt{obj}}$ for each $r \in \mathcal{V}_{ref}$.

Abstract heap environment transformers for typical pointer assignment statements are shown in Table 3, where $\lceil t, l \rceil \in \texttt{Obj}$, and for $r, r' \in \mathcal{V}_{diref}$, $o \in \texttt{Obj}$

$$(\texttt{henv} \bullet [r \mapsto o])r' = \begin{cases} \{o\} & \text{if } r = r' \notin [\![\texttt{Obj}]\!] \\ \texttt{henv}(r') \cup \{o\} & \text{if } r = r' \in [\![\texttt{Obj}]\!] \\ \texttt{henv}(r') & \text{otherwise} \end{cases}$$

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 pathes. 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 conflicted dataflow, and combining weights along the control flow will result in the weight **0**.

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

ExpFun	::=	$\lambda\texttt{henv}.$ ExpHenv
ExpHenv	::=	$\texttt{henv} \mid \text{ExpHenv} \bullet \text{ExpMap}$
ExpMap	::=	$[\text{Expf} \mapsto \text{Expt}]$
Expf	::=	$v \mid \text{Expt}.f \mid \text{Arrf}$
Expt	::=	$\lceil t, l \rceil \mid \texttt{henv}(v) \mid \texttt{henv}(\text{Expt}.f) \mid \text{Arrt}$
Arrf	::=	$[\![\text{Expt}]\!]$
Arrt	::=	$\texttt{henv}([\![\text{Expt}]\!])$

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 $(s, t \uparrow m)$ a path constraint $(s, t, m) \in \texttt{PathCons}$, 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** $\texttt{PathCons} \subseteq \mathcal{V} \times \mathcal{T} \times \mathcal{M}$, where $\mathcal{V} ::= v \mid \mathcal{V}.f \mid [\![\mathcal{V}]\!]$ is the set of references that syntactically allows nested field references and array structures.

Table 4
Abstract Heap Environment Transformer with Path Constraints

$x.m(r_0, ..., r_l)$	$(\tau_0 \circ \tau_1 \circ ... \circ \tau_k,\ \{(x, t \uparrow m)\})$ where $\tau_1, ..., \tau_k \in \texttt{Fun}$, $\texttt{Fun} = \{\lambda\texttt{henv.henv} \bullet [\texttt{arg}_i \mapsto \texttt{henv}(r_i)] \mid$ $r_i (0 \leq i \leq l)$ are arguments of reference type $\}$ $\tau_0 = \lambda\texttt{henv.henv} \bullet [\texttt{this} \mapsto \texttt{henv}(x)]$

The evaluation function $eval^{abs}$ is extended as $eval^{abs} : \text{Henv}^{abs} \to \mathcal{V} \cup \texttt{Obj} \to \mathcal{P}(\texttt{Obj})$, such that for $\texttt{henv} \in \text{Henv}^{abs}$, $o \in \texttt{Obj}$

$$\texttt{eval}^{abs}(\texttt{henv}, o) = \{o\} \qquad \texttt{eval}^{abs}(\texttt{henv}, v) = \texttt{henv}(v)$$

$$\texttt{eval}^{abs}(\texttt{henv}, \mathcal{V}.f) = \{\texttt{henv}(o.f) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, \mathcal{V})\}$$

$$\texttt{eval}^{abs}(\texttt{henv}, [\![\mathcal{V}]\!]) = \{\texttt{henv}([\![o]\!]) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, \mathcal{V})\}$$

Definition 4.3 Define $\texttt{Ref}$: ExpFun $\to \mathcal{V} \to \mathcal{P}(\text{Expt})$ such that for $\tau = \lambda\texttt{henv}.$ ExpHenv $\bullet$ $[\texttt{vf} \mapsto \texttt{vt}]$ and $\tau' = \lambda\texttt{henv}.$ ExpHenv $\in$ ExpFun,

$$\texttt{Ref}(\tau, v) = \begin{cases} \{\texttt{vt}\} & \text{if } \texttt{vf} = v \notin \text{Arrf} \\ \texttt{Ref}(\tau', v) \cup \{\texttt{vt}\} & \text{if } \texttt{vf} = v \in \text{Arrf} \\ \texttt{Ref}(\tau', v) & \text{otherwise} \end{cases}$$

$$\texttt{Ref}(\tau, \mathcal{V}.f) = \{\texttt{Ref}(\tau, \texttt{vt}'.f) \mid \texttt{vt}' \in \texttt{Ref}(\tau, \mathcal{V})\}$$

$$\texttt{Ref}(\tau, [\![\mathcal{V}]\!]) = \{\texttt{Ref}(\tau, [\![\texttt{vt}']\!]) \mid \texttt{vt}' \in \texttt{Ref}(\tau, \mathcal{V})\}$$

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 $\emptyset$ initially.

Definition 4.4 Define $\texttt{Ref}^{-1} : \mathcal{P}(\text{Expt}) \to \mathcal{P}(\mathcal{V} \cup \texttt{Obj})$ such that $\texttt{Ref}^{-1}(\mathbb{V}) = \bigcup_{\texttt{vt} \in \mathbb{V}} \texttt{Ref}^{-1}(\{\texttt{vt}\})$ for $\mathbb{V} \subseteq \text{Expt}$, where

$$\texttt{Ref}^{-1}(\{\lceil t, l \rceil\}) = \{\lceil t, l \rceil\}$$

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

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

$$\texttt{Ref}^{-1}(\{\texttt{henv}([\![\text{Expt}]\!])\}) = \{[\![\texttt{Ref}^{-1}(\{\text{Expt}\})]\!]\}$$

Definition 4.5 Define $\texttt{trace}$: ExpFun $\to \mathcal{V} \to \mathcal{P}(\mathcal{V} \cup \texttt{Obj})$ such that $\texttt{trace} = \texttt{Ref}^{-1} \circ \texttt{Ref}$.

Example 4.6 Let $\tau = \lambda\texttt{henv.henv} \bullet [x \mapsto \texttt{henv}(y)] \bullet [\texttt{henv}(z).f \mapsto o]$. Then $\texttt{trace}(\tau, x.f) = y.f$, $\texttt{trace}(\tau, z.f) = o$, and $\texttt{trace}(\tau, y) = y$.

Definition 4.7 Let $c \subseteq \texttt{PathCons}$ and $\tau \in$ Expfun. For $(s, t \uparrow m) \in c$,

$$\texttt{judge}(c, \tau) = \begin{cases} \texttt{error} & \text{if there exits } (s, t \uparrow m) \in c \text{ s.t.} \\ & \texttt{judge}(\{(s, t \uparrow m)\}, \tau) = \texttt{error} \\ \bigcup_{(s,t\uparrow m) \in c} \texttt{judge}(\{(s, t \uparrow m)\}, \tau) & \text{otherwise} \end{cases}$$

$$\texttt{judge}(\{(s, t \uparrow m)\}, \tau)$$
$$= \begin{cases} \texttt{error} & \text{if } \texttt{trace}(\tau, s) \subseteq \texttt{Obj} \text{ and for all } o \in \texttt{trace}(\tau, s), \\ & \quad \texttt{type}(o) \texttt{ conflicts with } t \text{ w.r.t. } m \\ \phi & \text{if there exists } o \in \texttt{trace}(\tau, s) \text{ for } o \in \texttt{Obj} \text{ and} \\ & \quad \texttt{type}(o) \text{ is } \texttt{compatible} \text{ with } t \text{ w.r.t. } m \\ \bigcup_{s' \in \texttt{trace}(\tau, s)} \{(s', t \uparrow m)\} & \text{otherwise} \end{cases}$$

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 known not to satisfy 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 $S_e = (D_e, \oplus_e, \otimes_e, 0_e, 1_e)$, such that

- $D_e = \mathcal{P}(\mathbb{D})$, where $\mathbb{D} = \{(f, c) \mid f \in ExpFun,\ c \subseteq \texttt{PathCons}\}$
- $0_e = \emptyset$ and $1_e = \{(\lambda\texttt{henv.henv}, \emptyset)\}$
- $w_1 \otimes_e w_2 = \{\ p_1 \otimes p_2 \mid p_1 = (\texttt{func}_1, c_1) \in w_1, p_2 = (\texttt{func}_2, c_2) \in w_2\ \}$

$$(\texttt{func}_1, c_1) \otimes (\texttt{func}_2, c_2) = \begin{cases} 0_e & \text{if } \texttt{jpc} = \texttt{error} \\ (\texttt{func}_2 \circ \texttt{func}_1, c_1 \cup \texttt{jpc}) & \text{otherwise} \end{cases}$$

  where $\texttt{jpc} = \texttt{judge}(c_2, \texttt{func}_1)$, $w_1, w_2 \in D_e$.
- $w_1 \oplus_e w_2 = w_1 \cup w_2$ for $w_1, w_2 \in D_e$

Remark 4.9 Both the associativity of $\otimes$ 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., $\top_{\texttt{obj}}$), as illustrated in Example 4.10. As a result, $(w_0 \otimes w_1) \otimes w_2 \sqsubseteq w_0 \otimes (w_1 \otimes w_2)$ for $w_0, w_1, w_2 \in \mathbb{D}$.

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 $\lambda\texttt{henv.henv} \bullet [\texttt{henv}(x).f \mapsto \texttt{henv}(w)] \bullet [y \mapsto \texttt{henv}(w)] \bullet [z \mapsto \texttt{henv}(\texttt{henv}(w).g)]$ by $(w_0 \otimes w_1) \otimes w_2$, and $\lambda\texttt{henv.henv} \bullet [\texttt{henv}(x).f \mapsto \texttt{henv}(w)] \bullet [y \mapsto \texttt{henv}(w)] \bullet [z \mapsto \top_{\texttt{obj}}]$ by $w_0 \otimes (w_1 \otimes w_2)$.

Remark 4.11 The points-to analysis presented above is flow-sensitive. It is easy to obtain parameterized flow-sensitivity as a hierarchy by loosening 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, $\bullet$ is reinterpreted as the

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 $\otimes_e$ operation on weights w_1, w_2 is extended as $w_1 \otimes_e w_2 = \{\lambda \texttt{henv}.\texttt{henv} \bullet p_1 \otimes p_2(\texttt{henv}) \bullet p_2 \otimes p_1(\texttt{henv}) \mid p_1 \in w_1, p_2 \in w_2\}$.

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

$$\texttt{pta}(r, C) = eval^{abs}(\delta(c, C)(\texttt{h}_{init}), r)$$

where $\delta(c, C)$ is from Definition 2.5 with $c = \langle \cdot, l_0 \rangle$ and $C = \langle \cdot, l.(\texttt{Ret})^* \rangle$.

We take $C = \langle \cdot, l.(\texttt{Ret})^* \rangle$ to represent all possible pushdown configurations as an approximation, when l is the top-most stack symbol. Therefore, $\texttt{pta}$ 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 sound (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 $\texttt{type} : \mathcal{O} \to \mathcal{T}$ and $\texttt{loc} : \mathcal{O} \to \texttt{Loc}$ be functions that return the type and the allocation site of a heap object, respectively. The **abstraction on heap objects** $\alpha : \mathcal{O} \to \texttt{Obj}$ is defined as follows,

- $\alpha(o) = (t, l)$ for $o \in \mathcal{O} \setminus \{\top_o, \bot_o\}$, $t = \texttt{type}(o) \in \mathcal{T}$, $l = \texttt{loc}(o) \in \texttt{Loc}$; and
- $\alpha(\top_o) = \top_{\texttt{obj}}$ and $\alpha(\bot_o) = \bot_{\texttt{obj}}$

The concretization is denoted by $\gamma = \alpha^{-1} : \texttt{Obj} \to \mathcal{P}(\mathcal{O})$. The powerset extensions of α and γ are denoted by $\alpha_o : \mathcal{P}(\mathcal{O}) \to \mathcal{P}(\texttt{Obj})$ and $\gamma_o : \mathcal{P}(\texttt{Obj}) \to \mathcal{P}(\mathcal{O})$.

Definition 4.14 For the program entry l_0 and the program point $l \in \texttt{Loc} \cup \texttt{Ret}$, let $\langle l_0, \texttt{h}_{init} \rangle \to^* \langle l, \texttt{henv} \rangle$. For $r \in \mathcal{V}_{ref}$ at l and $C = \langle \cdot, l.(\texttt{Ret})^* \rangle$, $\texttt{pta}(r, C)$ is *sound* if $\alpha_0(\texttt{eval}^{con}(\texttt{henv}, r)) \subseteq \texttt{pta}(r, C)$.

Definition 4.15 For abstract environment transformers f_1, $f_2 \in \text{ExpFun}$, $x \in \mathcal{V}_{ref}$ and $\texttt{henv} \in \text{Henv}^{abs}$, $f_1 \succcurlyeq f_2$ if $\texttt{eval}^{abs}(f_1(\texttt{henv}), x) \supseteq \texttt{eval}^{abs}(f_2(\texttt{henv}), x)$.

Theorem 4.16 *For a Jimple statement $s \in Stmt$, let f be the heap environment transformer of s, and f^{abs} be the abstract heap environment transformer of s. Then, $f^{abs} \succcurlyeq \alpha_o \circ f \circ \gamma_o$.*

Theorem 4.16 is proved by a case analysis on the Jimple statement s.

Lemma 4.17 *For $w_1, w_2 \in \mathbb{D}$, and $(\tau, c) \in w_1, (\tau', c') \in w_2$, $\texttt{henv} \in \texttt{Henv}^{abs}$, and $(s, t \uparrow m) \in c'$,*

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

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 "$\bullet \longrightarrow \bullet$" 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 $\tau(\texttt{henv})$ amounts to check that of all $s' \in \texttt{trace}(\tau, s)$ on `henv`.

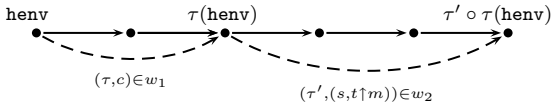


Fig. 2. Sound Tracing on Path Constrains

Theorem 4.18 *(**Soundness**) For $r \in \mathcal{V}_{ref}$ at $l \in \texttt{Loc} \cup \texttt{Ret}$ and $C = \langle \cdot, l.(\texttt{Ret})^* \rangle$, $\texttt{pta}(r, C)$ 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 points-to 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 type. For program statements whose execution does not change heap states, their corresponding heap environment transformers are thus identity function $\lambda\texttt{henv}.\texttt{henv}$, such as the conditional branching statement.

Definition 3.5 defines rules for judging whether a type t `conflicts` or `compatible` with a type 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 1.1. The analysis returns two abstract heap environment transformers. f_1 gives the precise dataflow summary of this program following the control flow "$4 \rightarrow 9 \rightarrow 15 \rightarrow 6 \rightarrow 9 \rightarrow 20$", which precisely infers that $\{o^3 \mapsto o^{15}, o^5 \mapsto o^{20}\}$. f_0 is a dataflow summary of the invalid call path "$6 \rightarrow 9 \rightarrow 15$" due to excluding the case (ii) above.

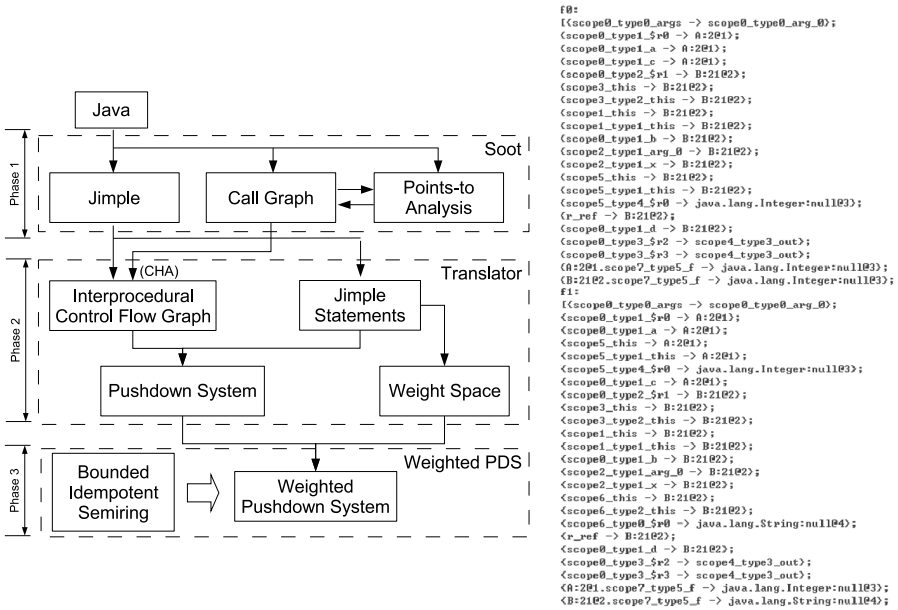


Fig. 3. The Prototype Framework and Running Profiles

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 $pta(y) \subseteq pta(x)$. 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

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.

SPARK[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 $\Theta(|\Delta| \cdot |\mathbb{D}| \cdot |T_{\oplus}| \cdot |T_{\otimes}|)$. $|\mathbb{D}|$ is the cardinality of the weight space. $|\Delta|$ is up to the program size by encoding. $|T_{\oplus}|$ and $|T_{\otimes}|$ 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 BBD package.

References

[1] D. A. Schmidt. Data flow analysis is model checking of abstract interpretation. *In Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38-48. ACM Press, 1998.

[2] Cousot, P. and Cousot, R., Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints, *Proc. 4th ACM Symposium on Principles of Programming Languages*, pp.238–252, 1977.

[3] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *the International Conference on Compiler Construction* (CC'03), pages 126-137, 2003.

[4] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *the 10^{th} International Static Analysis Symposium* (SAS), San Diego, CA, June 2003.

[5] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In the *15th International Conference on Compiler Construction* (CC 2006). LNCS volume 3923, Pages 47-64, 2006.

[6] Olin Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[7] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve. flow-insensitive pointer analysis. In *SIGPLAN 98 Conference on Programming Language Design and Implementation*, pages 97-105, June 1998.

[8] Reps, T., Schwoon, S., Jha, S., and Melski, D., Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming, 58(1C2):206–263, October 2005.

[9] Sagiv, M., Reps, T., and Horwitz, S., Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167 (1996), 131–170.

[10] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In the *8th International Conference on Concurrency Theory* (CONCUR'97), volume 1243 of LNCS, pages 135-150. Springer-Verlag, 1997.

[11] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701C726, November/December 1998.

[12] Jim Alves-Foss (Ed.). Formal syntax and semantics of Java. LNCS 1523 Springer 1999.

[13] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.

[14] M. Fändrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI),Montreal, Canada, June 1998.

[15] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), Snowbird, Utah, June 2001.

[16] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), Tampa Bay, Florida, October 2001.

[17] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2005.

[18] M. Sridharan and R. Bodik. Refinement-Based Context-Sensitive Points-To Analysis for Java. In the *Proceedings of ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation* (PLDI 2006).

[19] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. Proceedings of the *ACM SIGPLAN Conference on Programming Language Design and Implementation*(PLDI), 2004.

[20] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *the International Symposium on Software Testing and Analysis*, pages 1-11, 2002.

[21] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95) (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77-101.

[22] Bacon, D. F. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[23] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the 12th International Conference on Compiler Construction* (CC), pages 153-169, April 2003.

[24] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), San Diego, CA, June 2003.

[25] R. Vallee, Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay: Soot - a Java bytecode optimization framework, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research 1999 (CASCON '99), Ontario, Canada, November 1999.

[26] D. Beyer, A. Noack, C. Lewerentz. Efficient Relational Calculation for Software Analysis. IEEE Transactions on Software Engineering (TSE), 31(2):137-149, 2005.

Using CLP Simplifications to Improve Java Bytecode Termination Analysis

Fausto Spoto[1]

Dipartimento di Informatica
Università di Verona
Italy

Lunjin Lu[2]

Oakland University
USA

Fred Mesnard[3]

LIM IREMIA
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 efficiency bottleneck of the termination analyzer is the construction of a proof of termination for the generated constraint logic program, which is often very large in size. 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

[1] Email: fausto.spoto@univr.it
[2] Email: L2Lu@oakland.edu
[3] Email: frederic.mesnard@univ-reunion.fr

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

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 [8,10,7], functional programs [14] and term rewrite systems [11], whose semantics is relatively simple and well understood. More recently, termination analysis has been applied to imperative programs, dealing with primitive values only [9,15], lists [13,6,5,4] or any dynamic data-structure [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 [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 [17], a termination analysis is defined working for any sequential Java bytecode program [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 [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 [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 [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 `http://julia.scienze.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 4.1) 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 List<X> {
  private X head; private List<X> tail;
  public List(X[] values) { this(values,0); }
  public List(X h, List<X> t) { head = h; tail = t; }
  private List(X[] values, int l) {
    while (l < values.length && values[l] == null) l++;
    if (l < values.length) {
      this.head = values[l];
      if (l + 1 < values.length)
        this.tail = new List<X>(values,l + 1);
    }
  }
  public List<X> append(List<X> other) {
    if (tail == null) return new List<X>(head,other);
    else return new List<X>(head,tail.append(other));
  }
  public void afterInteger() { afterIntegerAux(false); }
  private void afterIntegerAux(boolean wasInteger) {
    if (head instanceof Integer) {
      if (tail != null) tail.afterIntegerAux(true);
    } else {
      if (tail != null) tail.afterIntegerAux(false);
      if (wasInteger) head = null;
    }
  }
  public String toString() {
    if (tail == null) return "* ";
    else if (head instanceof Integer) return "* " + tail.tail.toString();
    else return "* " + tail.toString();
  }
  public static void main(String[] args) {
    Object[] vs = { new Object(),3,3.14,null,new List<Integer>(3,null) };
    List<Object> list1 = new List<Object>(vs);
    List<Object> list2 = new List<Object>(vs);
    list2.afterInteger();
    String s = list1.append(list2).toString();
  }
}
```

Fig. 1. An example Java program.

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; 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 τ pseudo-bytecode creates the activation stack for a method with arguments of type

τ. 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 $c = \{\texttt{IS0} - \texttt{OS1} = 0, \texttt{IL1} - \texttt{OS4} = 0, \texttt{IS0} - \texttt{OS0} = 0, \texttt{IL1} - \texttt{OL1} = 0, \texttt{IL0} - \texttt{OL0} = 0, \texttt{OS3} \geq 0, \texttt{OS2} \geq 0, \texttt{IL0} - \texttt{OS3} \geq 1, \texttt{IL0} - \texttt{OS2} \geq 1\}$. The variables IS$n$ stand for the path-length of the nth stack element at the beginning of the block; OSn for their path-length at the end of the block; ILn and OLn are the same for the nth 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$i$ be a predicate expressing the path-length of the variables in scope at the beginning of block i. Its arity depends on which local variables and stack elements are in scope at the beginning of block i. We build clauses

$$\begin{array}{l} \texttt{block6391}(\texttt{IL0}, \texttt{IL1}, \texttt{IS0}) : -c, \texttt{block6392}(\texttt{OL0}, \texttt{OL1}, \texttt{OS0}, \texttt{OS1}, \texttt{OS2}, \texttt{OS3}, \texttt{OS4}). \\ \texttt{block6391}(\texttt{IL0}, \texttt{IL1}, \texttt{IS0}) : -c, \texttt{block6560}(\texttt{OL0}, \texttt{OL1}, \texttt{OS0}, \texttt{OS1}, \texttt{OS2}, \texttt{OS3}, \texttt{OS4}). \end{array} \quad (1)$$

since two arrows connect block 6391 with blocks 6392 and 6560. Two local variables L0 and L1 are in scope there (L0 implements this and L1 implements other). At the beginning of block 6391 there is only one stack element S0, 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 formalise 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$x$ or entry$x$. 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 $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o})$, where $\boldsymbol{i}$ and $\boldsymbol{o}$ are disjoint sequences of distinct variables and c is a linear integer constraint on $\boldsymbol{i}$ and $\boldsymbol{o}$. 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 iv and an output variable ov in the *CLP* program. The sequence $\boldsymbol{i}$ consists of only input variables and $\boldsymbol{o}$ 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.

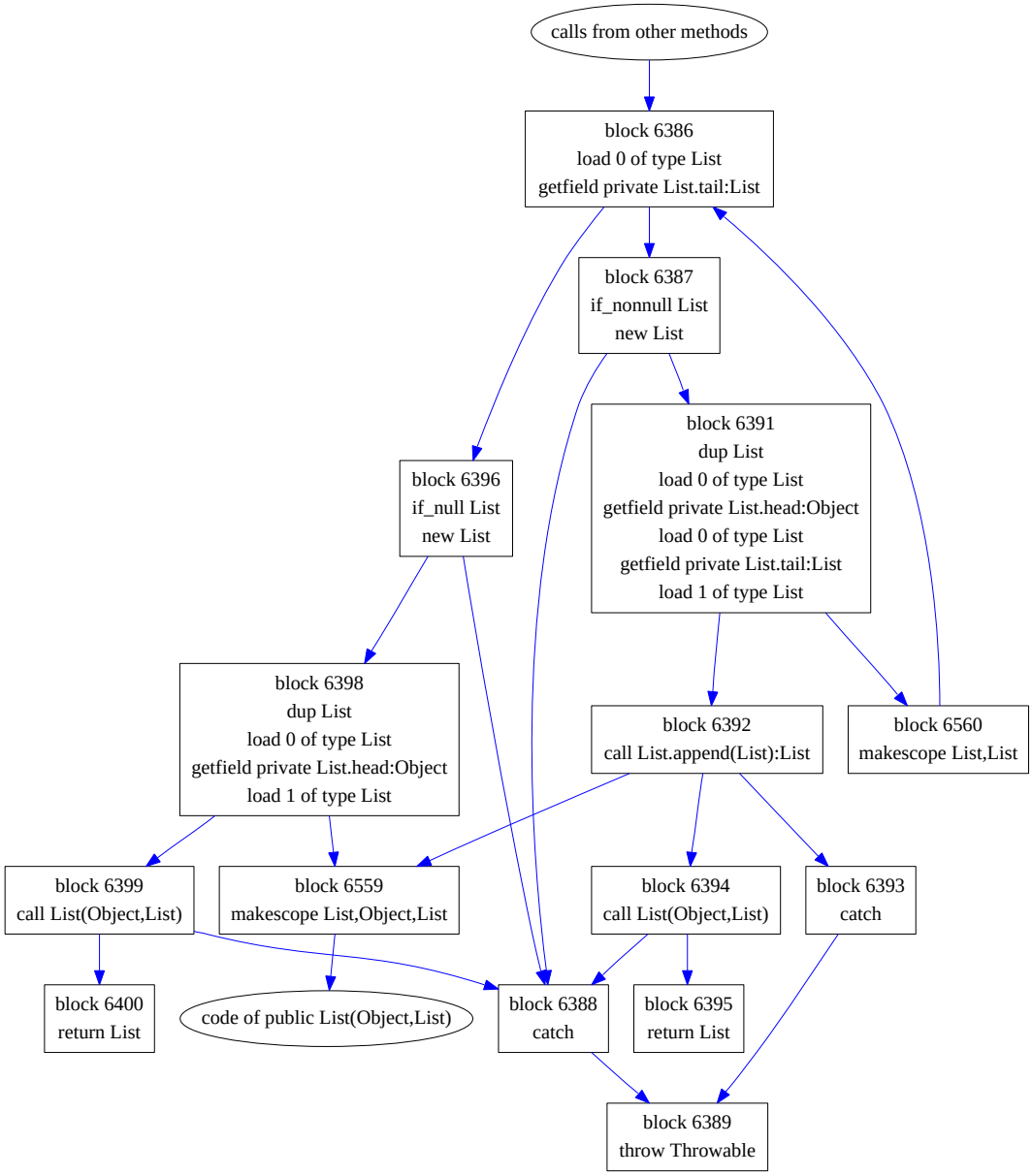


Fig. 2. The basic blocks for the method `append` in Figure 1.

Definition 3.1 [Valuation] A *valuation* θ is a map from a finite set of variables into integers. Let $\boldsymbol{v} = v_1 v_2 \cdots v_k$ be a sequence of variables and $\boldsymbol{val} = val_1 val_2 \cdots val_k \in \mathbb{Z}^k$. We write $[v_1 \mapsto val_1, \ldots, v_k \mapsto val_k]$ or $[\boldsymbol{v} \mapsto \boldsymbol{val}]$ for the valuation θ which is such that $\theta(v_i) = val_i$ for all $i = 1, \ldots, k$ and is undefined elsewhere. Let c be a constraint; then $c\theta$ is c where each variable v is replaced by $\theta(v)$. This notation is extended to any syntactical object, such as sequences of variables and predicates. The valuation θ is a *solution* of c if $c\theta$ is equivalent to true. Let p be a predicate; then $c[p(\boldsymbol{v}) \mapsto p(\boldsymbol{val})]$ stands for $c[\boldsymbol{v} \mapsto \boldsymbol{val}]$. □

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 $\boldsymbol{m}, \boldsymbol{n} \in \mathbb{Z}^*$. We say that $q(\boldsymbol{n})$ *is derived from* $p(\boldsymbol{m})$ using clause $C = (p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o}))$, written $p(\boldsymbol{m}) \rightarrow^C q(\boldsymbol{n})$, if there is a solution θ of $c[\boldsymbol{i} \mapsto \boldsymbol{m}]$ such that $q(\boldsymbol{n}) = q(\boldsymbol{o})\theta$. Clause C in $p(\boldsymbol{m}) \rightarrow^C q(\boldsymbol{n})$ is often omitted unless necessary. A *derivation* of $p_0(\boldsymbol{n}_0)$ is $p_0(\boldsymbol{n}_0) \rightarrow p_1(\boldsymbol{n}_1) \rightarrow \cdots \rightarrow p_k(\boldsymbol{n}_k)$ such that $p_{i+1}(\boldsymbol{n}_{i+1})$ is derived from $p_i(\boldsymbol{n}_i)$ for all $0 \leq i < k$. 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 $p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o})$, we 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 $q(\boldsymbol{n}) \text{ :- } c, s(\boldsymbol{m})$ with q and s in the same strongly-connected component (*i.e.,* in a loop) and also in a clause $t(\boldsymbol{v}) \text{ :- } c, q(\boldsymbol{w})$ with q and t in different strongly-connected components. We assume that entries are named `entry`x. From now on, when we say that a predicate is an entry of a *CLP* program, we mean that its name is `entry`x for some x.

Definition 3.3 [Termination] An entry p *terminates* in a program P if, for every $\boldsymbol{n} \in \mathbb{Z}^*$, all resolutions of $p(\boldsymbol{n})$ by 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 $P_1 \sqsupseteq P_2$, 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 importamnt 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_s. It will always be the case that P and P_s are termination-equivalent.

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 $p(\boldsymbol{n})$:- $c, q(\boldsymbol{m})$ *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 P_s be the same program deprived of those clauses that do not occur in a loop. Then $P \equiv P_s$.* □

Proof. We have $P_s \sqsupseteq P$ since $P_s \subseteq P$. It remains to prove $P \sqsupseteq P_s$. Programs P and P_s have the same set of entries. Let q be an entry. If q terminates in P then it terminates in P_s 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 P_s are used by that derivation, so that q diverges in P_s. □

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 $p(\boldsymbol{m})$:- $c_1, q(\boldsymbol{n})$ and $q(\boldsymbol{v})$:- $c_2, s(\boldsymbol{w})$, we can *unfold* them into the clause $p(\boldsymbol{m})$:- $c_1 \wedge c_2 \wedge \boldsymbol{n} = \boldsymbol{v}, s(\boldsymbol{w})$ (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 $q(\boldsymbol{n})$:- $c, q(\boldsymbol{m})$. Let P_s be P where q has been unfolded away. Then $P \equiv P_s$.* □

Proof. Programs P and P_s have the same set of entries. Let p be an entry of P_s. If p diverges in P_s then there is an infinite derivation d for p in P_s. Some steps of this derivation might use clauses derived from unfolding $r(\boldsymbol{m})$:- $c_1, q(\boldsymbol{n})$ with $q(\boldsymbol{v})$:- $c_2, s(\boldsymbol{w})$. We can replace those steps in d with two steps using those two clauses instead. The result is an infinite derivation for p that uses clauses of P. Hence p diverges in P. Conversely, if p diverges in P then there is an infinite derivation d for p in P. If a clause such as $r(\boldsymbol{m})$:- $c_1, q(\boldsymbol{n})$ is used during that derivation, then the subsequent step must use a clause of the form $q(\boldsymbol{v})$:- $c_2, s(\boldsymbol{w})$. Hence those two steps can be merged in d into a unique step that uses the unfolded clause $r(\boldsymbol{m})$:- $c_1 \wedge c_2 \wedge \boldsymbol{n} = \boldsymbol{v}, s(\boldsymbol{w})$. The resulting infinite derivation does not refer to q anymore and uses clauses in P_s. Hence p diverges in P_s. □

Note that Proposition 4.2 does not allow us to unfold away the entries to loops, whose termination is used to tell if each given loop terminates.

If we apply this simplification to the CLP program obtained at the end of Subsection 4.1, the number of clauses decreases from 12 to 8 and the time needed to prove all the entries terminating goes down from 2.72 to 1.48 seconds (including the time for unfolding).

4.3 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 $C_1 = (entry1(ix) := ix = ox, q(ox))$, $C_2 = (q(ix)$:- $ix = ox+1, entry1(ox))$ and $C_3 = (q(ix)$:- $ix \geq ox, r(ox))$. 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_s be P deprived of unsupported clauses. Then $P \equiv P_s$.* □

Proof. *Any divergent resolution in P_s is also a divergent resolution in P since P_s has less clauses than P. Any divergent resolution in P is also a divergent resolution in P_s since a divergent resolution in P cannot use any unsupported clause, or otherwise it would be finite.* □

Another simplification consists in removing subsumed clauses (see also [8]). Let for instance $C_1 = (p(\boldsymbol{i})$:- $c_1, q(\boldsymbol{o}))$ and $C_2 = (p(\boldsymbol{i})$:- $c_2, q(\boldsymbol{o}))$. We say that C_2 *subsumes* C_1 iff $c_1 \models c_2$ (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

```
entry3899(IL0):-OL0 >= 0,IL0 - OL0 >= 1,IL0 >= 2,entry3899(OL0).
entry3899(IL0):-OL0 >= 0,IL0 - OL0 >= 2,entry3899(OL0).
```

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_s be P deprived of subsumed clauses. Then $P \equiv P_s$.* □

Proof. *Any divergent resolution in P_s is also a divergent resolution in P since P_s 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_s. To that purpose, we prove that if $C_1 = (p(\boldsymbol{i}) \text{ :- } c_1, q(\boldsymbol{o}))$ is subsumed by $C_2 = (p(\boldsymbol{i}) \text{ :- } c_2, q(\boldsymbol{o}))$ then $p(\boldsymbol{m}) \rightarrow^{C_1} q(\boldsymbol{n})$ implies $p(\boldsymbol{m}) \rightarrow^{C_2} q(\boldsymbol{n})$ for any $p, q, \boldsymbol{m}$ and $\boldsymbol{n}$, which entails that any derivation step using C_1 can be replicated by using C_2. Assume hence that $p(\boldsymbol{m}) \rightarrow^{C_1} q(\boldsymbol{n})$. Then there is a solution θ of $c_1[\boldsymbol{i} \mapsto \boldsymbol{m}]$ such that $\boldsymbol{n} = \boldsymbol{o}\theta$. Since $c_1 \models c_2$, θ is also a solution of $c_2[\boldsymbol{i} \mapsto \boldsymbol{m}]$ and hence $p(\boldsymbol{m}) \rightarrow^{C_2} q(\boldsymbol{n})$.* □

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 $c^v = \exists_{-\{iv,ov\}}.c$ and $c^{-v} = \exists_{\{iv,ov\}}.c$. The constraint c^v is the *v-dedicated part* of c since it constrains variables iv and ov only; the constraint c^{-v} is the *v-independent part* of c since it does not constrain iv nor ov but only the other variables. Let us define an operation that removes a variable from a predicate, thus reducing its arity:

$$p(iv_1, \ldots, iv_n) \ominus v = \begin{cases} p(iv_1, \ldots, iv_{i-1}, iv_{i+1}, \ldots, iv_n) & \text{if } v \equiv v_i \\ p(iv_1, \ldots, iv_n) & \text{otherwise.} \end{cases}$$

Let us define $p(ov_1, \ldots, ov_n) \ominus v$ similarly. The transformation

$$Comp^{-v} = \{p(\boldsymbol{i}) \ominus v \text{ :- } c^{-v}, q(\boldsymbol{o}) \ominus v \mid p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o}) \in Comp\}$$

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_0 be an entry diverging in $Comp$. Then p_0 also diverges in $Comp^{-v}$.* □

Proof. *Since p_0 diverges in $Comp$, there is an infinite resolution*

$$p_0(\boldsymbol{n}_0) \rightarrow p_1(\boldsymbol{n}_1) \rightarrow p_2(\boldsymbol{n}_2) \rightarrow \cdots \rightarrow p_k(\boldsymbol{n}_k) \rightarrow \cdots$$

with $p_j(\boldsymbol{i}_j) \text{ :- } c_j, p_{j+1}(\boldsymbol{o}_j) \in Comp$, $p_{j+1}(\boldsymbol{n}_{j+1}) = p_{j+1}(\boldsymbol{o}_j)\theta_j$ and θ_j solution of $c_j[\boldsymbol{i}_j \mapsto \boldsymbol{n}_j]$. Hence $p_j(\boldsymbol{i}_j) \ominus v \text{ :- } c_j^{-v}, p_{j+1}(\boldsymbol{o}_j) \ominus v \in Comp^{-v}$ and θ_j is a solution of $c_j^{-v}[\boldsymbol{i}_j \mapsto \boldsymbol{n}_j]$ since $c_j \models c_j^{-v}$. Then θ_j is a solution of $c_j^{-v}[p_j(\boldsymbol{i}_j) \ominus v \mapsto p_j(\boldsymbol{n}_j) \ominus v]$

since c_j^{-v} *is* v*-independent. Thus,*

$$(p_{j+1}(\boldsymbol{o}_j) \ominus v)\theta_j = p_{j+1}(\boldsymbol{o}_j)\theta_j \ominus v = p_{j+1}(\boldsymbol{n}_{j+1}) \ominus v$$

and we can build the following infinite resolution of $p_0(\boldsymbol{n}_0) \ominus v$ *in* $Comp^{-v}$

$$p_0(\boldsymbol{n}_0) \ominus v \rightarrow p_1(\boldsymbol{n}_1) \ominus v \rightarrow p_2(\boldsymbol{n}_2) \ominus v \rightarrow \cdots \rightarrow p_k(\boldsymbol{n}_k) \ominus v \rightarrow \cdots$$

so that p_0 *diverges in* $Comp^{-v}$. □

In general, $Comp$ is not termination-equivalent to $Comp^{-v}$.

Example 4.8 Consider the strongly-connected component

$$Comp = \left\{ \begin{array}{c} entry1(ix, iy) \text{ :- } ix \geq 0, oy = ix, ox = iy, q(ox, oy) \\ q(ix, iy) \text{ :- } ox = iy - 1, oy = ix, entry1(ox, oy) \end{array} \right\}$$

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

$$Comp^{-x} = \left\{ \begin{array}{c} entry1(iy) \text{ :- } true, q(oy) \\ q(iy) \text{ :- } true, entry1(oy) \end{array} \right\}$$

Now $entry1$ does not terminate in $Comp^{-x}$. □

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 $Comp$ if, for every clause $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in Comp$, we have $c = c^v \wedge c^{-v}$. □

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

$$Comp = \left\{ \begin{array}{c} entry1(ix, iy) \text{ :- } ix \geq 0, ox = ix, oy = iy - 1, q(ox, oy) \\ q(ix, iy) \text{ :- } ox = ix - 1, oy = iy, entry1(ox, oy) \end{array} \right\}$$

□

4.5 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 $Comp$ is *right-open* if, for every $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in Comp$, we have that c^v is either *true* or $iv = ov$, or $ov \geq const$, $ov = const$ or $ov \leq const$ (or equivalent) , where *const* is an integer constant. *Left-openness* is defined analogously by switching ov with iv in the definition of right-openness. □

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

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

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

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

`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 $p_0(\boldsymbol{n}_0)$ in a strongly-connected component *Comp* where v is right-open. Let $p(\boldsymbol{i}_j)$:- $c_j, q(\boldsymbol{o}_j)$ be the clause used at the j^{th} resolution step. If c_j^v is $iv = ov$ then the j^{th} step simply copies the value of v from p_j to p_{j+1}. Otherwise, the value of v in p_j is not related to that in p_{j+1}: any value satisfying the v-dedicated part c_j^v of c_j may be picked up for v in p_{j+1}; such a value exists always due to the limited form of c_j^v. 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 $Comp^{-v}$ 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 p_0 be a divergent entry in $Comp^{-v}$. Then there is $\boldsymbol{m}_0 \in \boldsymbol{Z}$ and an infinite resolution of $p_0(\boldsymbol{m}_0)$ in $Comp^{-v}$, which we write as*

$$d_0 \rightarrow^{C_0} d_1 \rightarrow^{C_1} d_2 \rightarrow^{C_2} \cdots \rightarrow d_\ell \rightarrow^{C_\ell} d_{\ell+1} \cdots$$

where every clause $p(\boldsymbol{i}) \ominus v$:- $c^{-v}, q(\boldsymbol{o}) \ominus v$ used in each portion d_ℓ, for $\ell \geq 0$, is obtained from a clause $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in Comp$ with $c^v = (iv = ov)$ and each C_ℓ is obtained from a clause $C'_\ell = (p_\ell(\boldsymbol{i}_\ell)$:- $c_\ell, q_\ell(\boldsymbol{o}_\ell)) \in Comp$ with c_ℓ^v different from $iv = ov$. Let $x_\ell \in \mathbb{Z}$ be such that, for every $\ell > 0$, $\{ov \mapsto x_{\ell+1}\}$ is a solution of c_ℓ^v (hence x_0 is completely free). Let $p(\boldsymbol{m})$ be a call in $Comp^{-v}$ and $x \in \mathbb{Z}$. Then we define $p(\boldsymbol{m}) \oplus_v [x]$ as the call in Comp obtained from $p(\boldsymbol{m})$ by putting x at the position for v in the predicate p of Comp. It suffices to prove that there is an infinite resolution of $p_0(\boldsymbol{m}_0) \oplus_v [x_0]$ in Comp. Assume that

$$d_\ell = (p_{\ell,0}(\boldsymbol{m}_{\ell,0}) \rightarrow \cdots p_{\ell,j}(\boldsymbol{m}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1}) \cdots \rightarrow p_{\ell,f_\ell}(\boldsymbol{m}_{\ell,f_\ell}))$$

with $p_{0,0} = p_0$ and $\boldsymbol{m}_{0,0} = \boldsymbol{m}_0$. Let $p_{\ell,j}(\boldsymbol{n}_{\ell,j}) = p_{\ell,j}(\boldsymbol{m}_{\ell,j}) \oplus_v [x_\ell]$ for each $0 \leq j \leq f_\ell$. Since $p_{\ell,j}(\boldsymbol{m}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1})$, there is $p_{\ell,j}(\boldsymbol{i})$:- $(iv = ov) \wedge c, p_{\ell,j+1}(\boldsymbol{o}) \in Comp$ such that $p_{\ell,j}(\boldsymbol{i}) \ominus v$:- $c, p_{\ell,j+1}(\boldsymbol{o}) \ominus v \in Comp^{-v}$ and there is a solution θ of $c[p_{\ell,j}(\boldsymbol{i}) \ominus v \mapsto p_{\ell,j}(\boldsymbol{m}_{\ell,j})]$ such that $p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1}) = (p_{\ell,j+1}(\boldsymbol{o}) \ominus v)\theta$. Since c is v-independent, $\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_\ell\}$ is a solution of $(iv = ov) \wedge c[\boldsymbol{i} \mapsto \boldsymbol{n}_{\ell,j}]$ and $(\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_\ell\})(p_{\ell,j+1}(\boldsymbol{o})) = (p_{\ell,j+1}(\boldsymbol{o}) \ominus v)\theta \oplus_v [x_\ell] = p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1}) \oplus_v$

$[x_\ell] = p_{\ell,j+1}(\boldsymbol{n}_{\ell,j+1})$. *Thus,* $p_{\ell,j}(\boldsymbol{n}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{n}_{\ell,j+1})$ *for* $0 \leq j \leq f_\ell - 1$ *and*

$$d'_\ell = (p_{\ell,0}(\boldsymbol{n}_{\ell,0}) \rightarrow \cdots p_{\ell,j}(\boldsymbol{n}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{n}_{\ell,j+1}) \cdots \rightarrow p_{\ell,f_\ell}(\boldsymbol{n}_{\ell,f_\ell}))$$

is a derivation in Comp. We show now that $p_{\ell,f_\ell}(\boldsymbol{n}_{\ell,f_\ell}) \rightarrow^{C'_\ell} p_{\ell+1,0}(\boldsymbol{n}_{\ell+1,0})$ *so that we obtain an infinite resolution* $d'_0 \rightarrow d'_1 \rightarrow \cdots d'_\ell \rightarrow d'_{\ell+1} \rightarrow \cdots$ *in Comp. Since* $p_{\ell,f_\ell}(\boldsymbol{m}_{\ell,f_\ell}) \rightarrow^{C_\ell} p_{\ell+1,0}(\boldsymbol{m}_{\ell+1,0})$*, we know that* $p_\ell = p_{\ell,f_\ell}$*,* $q_\ell = p_{\ell+1,0}$ *and there is a solution* θ *of* $c_\ell^{-v}[p_\ell(\boldsymbol{i}_\ell) \ominus v \mapsto \boldsymbol{m}_{\ell,f_\ell}]$ *such that* $q_\ell(\boldsymbol{m}_{\ell+1,0}) = (q_\ell(\boldsymbol{o}_\ell) \ominus v)\theta$*. Then* $\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_{\ell+1}\}$ *is a solution of* $c_\ell^{-v}[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}]$*, since* c_ℓ^{-v} *is* v*-independent, and it is also a solution of* $c_\ell^v[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}]$ *and hence of* $c_\ell[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}]$ *since* $c_\ell^v[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}] = c_\ell^v[iv \mapsto x_\ell] = c_\ell^v$ *and* c_ℓ^v *contains only* ov *and* $\{ov \mapsto x_{\ell+1}\}$ *is a solution of* c_ℓ^v*. Also,* $q_\ell(\boldsymbol{o}_\ell)(\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_{\ell+1}\}) = (q_\ell(\boldsymbol{o}_\ell) \ominus v)\theta \oplus_v [x_{\ell+1}] = q_\ell(\boldsymbol{m}_{\ell+1,0}) \oplus_v [x_{\ell+1}] = q_\ell(\boldsymbol{n}_{\ell+1,0})$*. Hence* $p_{\ell,f_\ell}(\boldsymbol{n}_{\ell,f_\ell}) \rightarrow^{C'_\ell} p_{\ell+1,0}(\boldsymbol{n}_{\ell+1,0})$*.* □

If we apply this simplification to the CLP program obtained at the end of Subsection 4.3, the number of clauses goes down from 7 to 6 (because of entailment checks) and there are less arguments in predicates. The time needed to prove all the entries terminating goes down from 1.25 to 1.02 seconds (including the time to apply all the simplifications discussed up to now).

4.6 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 $x \in \mathbb{Z}$ such that, for every $p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o}) \in Comp$, the valuation $\{iv \mapsto x, ov \mapsto x\}$ 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:

```
block3853(IL0,IL1,IL2):-IL2 - OL2 = -1,IL1 - OL1 = 0,IL0 - OL0 = 0,
                        IL1 - IL2 >= 1,block3853(OL0,OL1,OL2).
block3853(IL0,IL1,IL2):-IL2 - OL2 = -1,IL1 - OL1 = 0,OL0 = 1,
                        IL1 - IL2 >= 2,entry3849(OL0,OL1,OL2).
entry3849(IL0,IL1,IL2):-IL2 - OL2 = 0,IL1 - OL1 = 0,IL0 - OL0 = 0,
                        IL0 >= 1,block3853(OL0,OL1,OL2).
```

By taking $x = 1$, we conclude that `L0` is uniform. □

Example 4.16 Uniform variables and left- or right-open variables are different concepts. For instance, variable `L0` is uniform in the component of Example 4.15 but it is not left-open nor right-open. Conversely, variable x is left-open in the component

$$entry1(ix, iy) \text{ :- } iy \geq 0, ox = ix, ix \geq 3, oy = iy - 1, p(ox, oy)$$
$$p(ix, iy) \text{ :- } ox = ix, ix \leq 0, oy = iy, entry1(ox, oy)$$

but it is not uniform there. □

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 v 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. `JLex` 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 megabyte 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, unlocalized 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`, `JLex` and `Kitten`. In general, more simplifications means better efficiency. This relation is not always true. For instance, building a proof of termination for `JLex` 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.

program	meth.	original	4.1	4.2	4.3	4.5	4.6
`Ackermann`	5	7.11	0.21	0.21	0.21	0.21	0.21
precision		5	5	5	5	5	5
`BubbleSort`	5	19.07	1.55	0.71	0.71	0.49	0.49
precision		3	4	5	5	5	5
`NQueens`	222	-	210.31	156.32	92.29	47.77	34.34
precision		-	171	171	171	171	171
`JLex`	137	-	228.51	335.85	374.82	121.95	81.21
precision		-	84	87	102	102	102
`Kitten`	947	-	200.39	226.79	152.47	93.70	79.35
precision		-	811	827	827	827	827

Fig. 3. The termination analyses of some programs. Times are in seconds. The second line (*precision*), for each program, reports the number of methods proved to terminate. In the header, we refer to the subsection where the simplification is described.

6 Conclusion

We have presented techniques for simplifying the CLP programs that are automatically generated during termination analysis of Java bytecode programs. Those techniques are proved to keep the termination-equivalence of the CLP programs. Their application to some real case of analysis shows that they decrease the time for building a proof by some order of magnitude. Moreover, simplified CLP programs induce less timeouts during the construction of the proof of termination, so that our simplification techniques actually induce more precise termination analyses.

In [2], *useless variables* are eliminated from *CLP* programs expressing cost relationships for Java bytecode programs. That technique removes most stack variables. We have verified that almost no stack variable survives after our unfolding of clauses (Subsection 4.2). Our unfolding can be seen as a *CLP* view of the simplification done in [2] from a Java bytecode perspective. On the one hand, as in [2] the elimination of variables is done earlier, all related static analyses benefit from this simplification. On the other hand, note that we have a correctness proof for that simplification and that subsequent simplifications are not related to that in [2].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In R. L. Wainwright and H. Haddad, editors, *Proc. of the 2008 ACM Symposium on Applied Computing (SAC'08)*, pages 368–375, Fortaleza, Ceara, Brazil, March 2008. ACM.

[3] A. M. Ben-Amram and M. Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C. R. Ramakrishnan and J. Rehof, editors, *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 218–232, Budapest, Hungary, 2008. Springer.

[4] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance Analyses from Invariance Analyses. In M. Hofmann and M. Felleisen, editors, *Proc. of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 211–224, Nice, France, January 2007.

[5] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 386–400, Seattle, WA, USA, August 2006. Springer.

[6] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists Are Counter Automata. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531, Seattle, WA, USA, August 2006. Springer.

[7] M. Codish. Proving Termination with (Boolean) Satisfaction. In A. King, editor, *Proc. of the 17th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'07)*, volume 4915 of *Lecture Notes in Computer Science*, pages 1–7, Kongens Lyngby, Denmark, 2007. Springer.

[8] M. Codish and C. Taboch. A Semantics Basis for Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418, Seattle, WA, USA, August 2006. Springer.

[10] S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *Theory and Practice of Logic Programming (TPLP)*, 5(1-2):75–91, 2005.

[11] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In U. Furbach and N. Shankar, editors, *3th International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286, Seattle, WA, USA, August 2006. Springer.

[12] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[13] A. Loginov, T. W. Reps, and M. Sagiv. Refinement-Based Verification for Possibly-Cyclic Lists. In T. W. Reps, M. Sagiv, and J. Bauer, editors, *Proc. of Theory and Practice of Program Analysis and Compilation, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*, pages 247–272. Springer, 2006.

[14] P. Manolios and D. Vroon. Termination Analysis with Calling Context Graphs. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414, Seattle, WA, USA, August 2006. Springer.

[15] A. Podelski and A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), May 2007.

[16] F. Spoto, P. M. Hill, and É. Payet. Path-Length Analysis for Object-Oriented Programs. In *First International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*, Vienna, Austria, March 2006. Available at the web address `http://profs.sci.univr.it/~spoto/papers.html`.

[17] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode Based on Path-Length. Submitted for publication in September 2007. Available at the web address `http://profs.sci.univr.it/~spoto/papers.html`.

[18] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *London Mathematical Society*, 42(2):230–265, 1936.

The Non-Interference Protection in BML[1]

Aleksy Schubert[2] Daria Walukiewicz-Chrząszcz[3]

Institute of Informatics
University of Warsaw
ul. Banacha 2
02–097 Warsaw
Poland

Abstract

Many information-flow type systems have been developed that allow to control the non-interference of information between the levels of classification in the Bell-LaPadula model. We present here a translation of typing information collected for bytecode programs to a bytecode program logic. This translation uses the syntax of a bytecode specification language BML. A translation of this kind allows including the check of the non-interference property in a single, unified verification framework based on a program logic and thus can be exploited within a foundational proof-carrying code infrastructure. It also provides a flexible basis for various declassification strategies that may be useful in a particular code body.

Keywords: Java, bytecode, specification, BML, non-interference

1 Introduction

The application of formal specification methods at the level of the Java bytecode has several advantages. (1) This allows to provide descriptions and verify properties of programs written in the bytecode itself. (2) It allows to do a unified formalised development for languages other than Java, but compiling to the Java bytecode. In particular, it allows to conduct a unified formal verification in projects with several source code languages. (3) Proofs for bytecode programs may enable several optimisations in JIT compilers. (4) As bytecode is the language which is actually executed, it is possible to couple with programs their proof carrying-code (PCC) certificates. (5) Since Java programs are distributed in their bytecode version, it is possible for a software distributor to develop its own certificate to ensure a particular property its clients are interested in. These reasons led to a proposal of a bytecode program logic [7] and, based on this foundation, a specification language for the bytecode — Bytecode Modeling Language (BML) [11]. The latter language is based

[1] This work was partly supported by Polish government grant 177/6.PR UE/2006/7 and and Polish government grant 3 T11C 002 27.

[2] Email: alx@mimuw.edu.pl

[3] Email: daria@mimuw.edu.pl

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

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 [10].

We consider here the non-interference property of bytecode programs. Many type systems to 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 [4,5] 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 verifiable 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 Hoare-like logic available in BML 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 Amtoft et al [1]. Moreover, BML 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 BML and base the safety of the program on the type system soundness. Still another limitation of the approach based on BML is that the self-composition [3] cannot be expressed here (although it is available in MOBIUS 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.

[4] The translation from this paper does not reduce the trusted logical base to the one of the foundational PCC. To achieve that one has to link the resulting formulae with the non-interference property expressed in the foundational logic e.g. the property expressed in [8] for while programs.

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 $\mathrm{dom}(f)$ to denote the domain of the (partial) function f. Similarly, $\mathrm{rng}(f)$ is the image of f. We write $f : A \rightharpoonup B$ when f is a partial function from A to B. The powerset of A is $P(A)$. We write $\boldsymbol{k}$ to denote a vector of values. The notation $|\boldsymbol{k}|$ expresses the length of the vector and $\boldsymbol{k}(0), \ldots, \boldsymbol{k}(|\boldsymbol{k}|-1)$ are subsequent elements of the vector. The set of finite sequences over a set A is A^*.

Java bytecode programs and specifications A Java bytecode program P is a set of classes with one singled out method main_P. A class C is a set of fields and methods. Each field f has a name f_n and a type f_t. Similarly, a method m has a name $N(m)$, a signature T_m and a body B_m [5]. 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 $\mathcal{PP}_m$ (we omit the subscript m when it is clear from the context).

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

- a method specification $S_m = (R_m, T_m, \Phi_m)$ where R_m is the precondition of the method m, T_m is the postcondition of the method, and Φ_m is the method invariant which holds in each accessible state of the method;
- a local specification table G_m which assigns to each label in the method body B_m 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 Q_m which assigns to labels in B_m further assertions (this corresponds to the BML `assert` annotations);
- a local instruction table Ins_m which assigns to each label l in the method body B_m 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 $\mathsf{Q}_m(l)$ (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 statically the non-interference property. We also assume that the attacker

[5] The separation of the identities for the method and its name serves to model the inheritance.

is unable to observe the termination of the programs.

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

- A security level k_{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_{obs}).
- A policy function ft assigns to each field its security level. This allows us to express the non-interference property we are interested in.
- A policy function Γ that associates to each method identifier $N(m)$ and security level $k \in S$ a security signature $\Gamma_{N(m)}[k]$ [6]. This signature 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 $\mathsf{Policies}_\Gamma(m) = \{\Gamma_{N(m)}[k] \mid k \in \mathcal{S}\}$. The security signature has the shape $\boldsymbol{k_p} \xrightarrow{k_h} \boldsymbol{k_r}$:
  - The vector $\boldsymbol{k_p}$ describes the security levels appropriate for the local variables of the method (in particular it assigns also the levels to the input parameters), $k_p[0]$ is the upper bound on the security level of an object that calls the method.
  - The value k_h describes the lower bound in the security levels of the heap operations performed by the method.
  - The vector $\boldsymbol{k_r}$ describes the security levels for the method results (both normal and exceptional ones); it is a list of the form $\{n : k_n, e_1 : k_{e_1}, \ldots, e_n : k_{e_n}\}$, where k_n 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_{e_i} is the upper bound on the security level of the exception. We use the notation $k_r[n]$ and $k_r[e_i]$ for k_n and k_{e_i}.

Non-interference The non-interference property is articulated by a safety definition in [4,5]. Informally, a program is non-interferent if all its methods are safe; a method is safe if two terminating runs of the method with inputs that cannot be distinguished by an attacker, and equivalent heaps, yield results 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 $\mapsto^\tau \subseteq PP \times PP$ defined on the program points. This relation is parametrised by a tag τ since a instructions may have several successors as they may execute normally (the tag is $\emptyset$) or may trigger exceptions (the tag is the class of the exception). Intuitively, j is a successor of i ($i \mapsto j$) 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 $i \mapsto$ when $\mapsto$ is undefined for i i.e. if i corresponds to a return instruction (or $i \mapsto^\tau$ if i corresponds to an instruction that may throw an exception that is not handled locally). Note that an instruction may have more

[6] In [5] a less precise notation $\Gamma_m[k]$ is used.

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 $(\mathtt{region}, \mathtt{jun})$. 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 $\mathtt{region}$ function describes the internal parts of these regions while $\mathtt{jun}$ the connections between them. The types of the functions are the following:

$$\mathtt{region}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \to P(\mathcal{PP}) \qquad \mathtt{jun}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \rightharpoonup \mathcal{PP}$$

The functions can be axiomatised 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 [5]. In this type system, every method is checked against its signatures separately. The type system is parametrised by:

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

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

(i) $\mathsf{classAnalysis}$ which for a program point returns the set of exception classes of exceptions that may be thrown at the program point.

(ii) $\mathsf{excAnalysis}$ which for a method name $N(m)$ returns the set of exception classes that are possibly thrown by m.

(iii) nbLocs which for a method name $N(m)$ returns the number of its local variables.

(iv) nbArgs which for a method name $N(m)$ returns the number of its arguments.

(v) $\mathsf{Handler}_m$ 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, $\mathtt{region}_m$ and a signature sgn if there exists a security environment $\mathsf{se} : PP \to S$ and a function $\mathsf{st} : PP \to S^*$ such that $st(0) = \varepsilon$ and for all $i, j \in PP$, $e \in \{\emptyset\} \cup \mathcal{C}$:

(1) if $i \mapsto^e j$ then there exists $s \in S^*$ such that $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies s$ and $s \sqsubseteq \mathsf{st}(j)$,

(2) if $i \mapsto^e$ then $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies$

where $\sqsubseteq$ 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 $\ldots \vdash^e \ldots \implies \ldots$, for $\mathtt{ifeq}$, is given in Figure 1 [7]. The set of all typing rules is presented in [4].

A program P is *typable* with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr satisfying SOAP if every method m from P is typable with respect to ft, Γ, $\mathtt{region}_m$ and all signatures in $\mathsf{Policies}_\Gamma(m)$.

[7] We limit our exposition to the $\mathtt{ifeq}$ instruction due to the space restrictions.

$$\frac{P_m[i] = \texttt{ifeq}\ j \qquad \forall j' \in \texttt{region}(i, \emptyset),\ k \leq se(j')}{\Gamma, \mathsf{ft}, \texttt{region}, \mathsf{se}, \boldsymbol{k_p} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k :: st \implies \texttt{lift}_k(st)}$$

Figure 1. The typing rule for `ifeq`

Intuitively, the function 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 st 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 st; at last sgn is the currently used security signature for the analysed method.

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

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

The main theorem of [5] states that typable programs are non-interferent.

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

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 ==>. We also use the BML syntax to access and initialise 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 variable declaration; bound on the quantification; actual formula)

where the *variable declaration* has the same form as a variable declaration in Java and introduces the quantified variables. The *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 *actual formula* is the formula we are interested in. A *bound on the quantification* B and an *actual formula* A are understood as the implication $B ==> A$.

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 Γ of method signatures,
- a global policy ft that provides security levels of fields,
- a cdr structure $(\texttt{region}_m, \texttt{jun}_m)$[8],
- 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 $\mathsf{se} : PP \to S$, and a function $\mathsf{st} : PP \to S^*$.

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

- maxEx the number of all exception types in P.
- maxS the maximal security level used to type-check P; the level $0 \notin S$ will be used to mark the undefined value. Let $S0 = S \cup \{0\}$.
- maxNbArg the maximal number of arguments of all methods in P.

[8] We actually do not provide the definition for $\texttt{jun}_m$ as the function does not occur in the typing rules in [4].

- lm 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 S gft_f = ft(f);
```

this enables access to the security levels of `f`. The domain of Γ, excAnalysis, nbLocs, nbArgs 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 $0, \ldots,$ 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 values/functions.

```
public static final ghost int gnbArgs_N(m) = nbArgs(N(m));
public static final ghost int gnbLocs_N(m) = nbLocs(N(m));
public static final ghost boolean [maxEx] gexcAnalysis_N(m) =
                          { e_0, ..., e_maxEx-1 };
public static final ghost S0
 [maxS][gnbLocs_N(m)+3+maxEx] gsgn_N(m) =
   { { s_0,0, ..., s_0,gnbLocs_N(m)+3+maxEx-1 }, ...
     { s_maxS-1,0, ..., s_maxS-1,gnbLocs_N(m)+3+maxEx-1 } };
```

The last two definitions make use of additional values defined below. The information contained in **gexcAnalysis**$\_N(m)$ is defined with the use of:

$$e_i = \begin{cases} \textbf{true} & \text{when excAnalysis}(N(m)) \text{ says that the exception } i \text{ is thrown in } m, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

The security signature $\Gamma_m[i] = \boldsymbol{k_p} \xrightarrow{k_h} \boldsymbol{k_r}$ allows us to give the values for $s_{i,j}$:

$$s_{i,j} = \begin{cases} \boldsymbol{k_p}(j) & j < |\boldsymbol{k_p}|, \\ k_h & j = |\boldsymbol{k_p}|, \\ \boldsymbol{k_r}(j - |\boldsymbol{k_p}| - 1) & j > |\boldsymbol{k_p}|. \end{cases}$$

This definition allows us to explain the meaning of gsgn$[i][j]$ in the following way. For a given security level i, gsgn$[i][0]$ is the security level of the object that calls the method m, gsgn$[k][1 \ldots gnbLocs_N(m)]$ are security levels of parameters and local variables (note that nbLocs$(N(m)) = |\boldsymbol{k_p}| - 1$), the value gsgn$[k][gnbLocs_N(m)+1]$ is the level of heap operations, the value gsgn$[k][gnbLocs_N(m) + 2]$ is the level of a normal return value and

$$\text{gsgn}[k][gnbLocs\_N(m) + 3 \ldots gnbLocs\_N(m) + 3 + \text{maxEx} - 1]$$

are the security levels in which corresponding exceptions might be propagated (note that maxEx $\geq |\boldsymbol{k_r}| - 1$).

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

```
ghost boolean [lm][maxEx] gclassAnalysis =
  { { c_{0,0}, ..., c_{0,maxEx-1} }, ..., { c_{lm-1,0}, ..., c_{lm-1,maxEx-1} } };
```

where

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

```
ghost boolean [lm][maxEx+1][lm] gregion = {
    { { r_{0,0,0}, ..., r_{0,0,lm-1} }, ..., { r_{0,maxEx,0}, ..., r_{0,maxEx,lm-1} } },  ...,
    { { r_{lm-1,0,0}, ..., r_{lm-1,0,lm-1} }, ..., { r_{lm-1,maxEx,0}, ..., r_{lm-1,maxEx,lm-1} } }
};
```

where

$$r_{i,j,k} = \begin{cases} \textbf{true} \text{ when } k \in \mathtt{region}_m(i,e) \text{ and the exception corresponding to } e \text{ is } j, \\ \textbf{true} \text{ when } k \in \mathtt{region}_m(i,\emptyset) \text{ and } j = \mathsf{maxEx}, \\ \textbf{false} \text{ otherwise.} \end{cases}$$

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

```
ghost int [lm][maxEx] gHandler =
  { { h_{0,0}, ..., h_{0,maxEx-1} }, ..., { h_{lm-1,0}, ..., h_{lm-1,maxEx-1} } };
```

where $h_{i,j} = \mathsf{Handler}_m(i,e)$ 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_i and st_i for $i \in S$ to refer to the elements of the indexed families.

```
ghost S [maxS][lm] gse =
  { { v_{0,0}, ..., v_{0,lm-1} }, ..., { v_{maxS-1,0}, ..., v_{maxS-1,lm-1} } };
```

where $v_{i,j} = \mathsf{se}_i(j)$.

```
ghost S0 [maxS][lm][maxStack] gst = {
    { { t_{0,0,0}, ..., t_{0,0,maxStack-1} }, ..., { t_{0,lm-1,0}, ..., t_{0,lm-1,maxStack-1} } },
    ...,
    { { t_{maxS-1,0,0}, ..., t_{maxS-1,0,maxStack-1} }, ...,
      { t_{maxS-1,lm-1,0}, ..., t_{maxS-1,lm-1,maxStack-1} } }
};
```

where $t_{i,j,k} = n$ whenever $(k+1)$-st element of the sequence $\mathsf{st}_i(j)$ is n. Note that the function st_i gives security levels for the stack positions so that the length of each $\mathsf{st}_i(j)$ is less that the maximal stack length maxStack. We also assume that for each i, j the elements of $gst[i][0][j]$ 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 typability property from Def. 1. We do it for each method m 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 $Q_i(s0, s)$. If the point i in m is annotated with Q_m then $Q_i(s0, s)$ is supposed to hold in every state s at i during any execution of m with the initial state $s0$ satisfying $R_m(s0)$ (i.e. the precondition of the method). Intuitively, $Q_i(s0, s)$ 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

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 $i \mapsto^e j$ (or $i \mapsto^e$) some properties hold. For every i these properties are expressed by formulae $N(i)(s)$. We define QNI_m, the local annotation table extended with the non-interference checking, as

$$\begin{aligned}\mathsf{QNI}_m(i) = \lambda c0 \in \mathsf{State}\ \ \lambda c \in \mathsf{State}.\\ Q_i(c0, c)\ \&\&\ N(i)(1)\ \&\&\ \ldots\ \&\&\ N(i)(\mathsf{maxS}).\end{aligned} \tag{1}$$

This formula expresses a new assert before the instruction at i which 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 obeyed.

The security guarding formulae $N(i)(s)$ have similar form; it is

$$\begin{aligned}&(\backslash\mathsf{forall}\ \mathsf{int}\ e,\ j; 0 \le e\ \&\&\ e \le \mathsf{maxEx}\ \&\&\ 0 \le j\ \&\&\ j < \mathsf{lm};\\ &\qquad (i \mapsto^e j \Longrightarrow (\mathsf{Reg}_1^{\mathtt{inst}(i),s}(\boldsymbol{p_1})\ ||\ \ldots\ ||\ \mathsf{Reg}_k^{\mathtt{inst}(i),s}(\boldsymbol{p_k}))) \ \&\& \\ &\qquad (i \mapsto^e \Longrightarrow (\mathsf{Reg}_{k+1}^{\mathtt{inst}(i),s}(\boldsymbol{p_1})\ ||\ \ldots\ ||\ \mathsf{Reg}_{k'}^{\mathtt{inst}(i),s}(\boldsymbol{p_{k'}}))))\end{aligned} \tag{2}$$

[9]

where $\mathtt{inst}(i)$ is the instruction at the label i in the method body m. Note also that $i \mapsto^e j$ (as well as $i \mapsto^e$) is 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 $j == i + 1\ \&\&\ e == \mathsf{maxEx}$ [10]. Here, the condition that maxEx equals e enforces that we consider a normal step. In the case when the method has an exception handler of e_0 at the point j' we define $i \mapsto^e j$ to be $j == j'\ \&\&\ e == e_0$. As the `ifeq` j_0 has two successors, but one rule handles the instruction in the type system, the premise of the implication is $(j == i + 1\ ||\ j == j_0)\ \&\&\ e == \mathsf{maxEx}$ in this case.

Every $\mathsf{Reg}_i^{\mathtt{inst}(i),s}(\boldsymbol{p_i})$ for $i = 1, \ldots, k$ or for $i = k + 1, \ldots, k'$ corresponds to one of possibly applicable typing rules for instruction $\mathtt{inst}(i)$ in case $i \mapsto^e j$ (or $i \mapsto^e$). The type system considers an instruction to be correct when at least one of the rules can be successfully satisfied. Therefore, the formulae $\mathsf{Reg}_i^{\mathtt{inst}(i),s}$ are combined as an alternative. Let us point out that the vectors $\boldsymbol{p_i}$ are parameters of the instruction `inst`. For instance, there is one rule for `ifeq` and $\mathsf{Reg}_1^{\mathtt{ifeq},s}(j)$, corresponding to

[9] We add this subformula only in case the instruction may throw an exception.

[10] Note that the addition can be performed 'on-the-fly' in the course of the translation and therefore is not a part of the formula syntax.

`ifeq` j, equals to

$$
\begin{array}{ll}
(\backslash\text{forall int } j';\ 0 \le j' \ \&\&\ j' < \mathsf{lm}; & \\
\qquad gregion[i][\mathsf{maxEx}][j'] ==> gst[s][i][\mathsf{cntr}] \le gse[s][j']) & \&\& \\
(\backslash\text{forall int } p;\ 0 \le p \ \&\&\ p < \mathsf{cntr}; & \\
\qquad gst[s][i][p] \sqcup gst[s][i][\mathsf{cntr}] \le gst[s][j][p]) & \&\& \\
(\backslash\text{forall int } p;\ \mathsf{cntr} \le p \ \&\&\ p \le \mathsf{maxStack}; & \\
\qquad gst[s][j][p] = 0) &
\end{array}
\tag{3}
$$

We can now relate the formula to the rule in Figure 1. First, observe, that the index cntr is a counter of the operand stack; hence $gst[s][i][\mathsf{cntr}]$ points to the top of the stack and it corresponds to k from the rule. The first precondition of the rule in Figure 1 holds as the formula is generated only for `ifeq` instruction. The formula above consists of three $\forall$ subformulae. The first subformula expresses the condition $\forall j' \in \mathtt{region}(i, \emptyset),\ k \le se(j')$ from the precondition in the rule in Figure 1. Recall that maxEx value in second parameter of *gregion* means a normal (non-exceptional) behaviour. The second and the third subformulae state that $\mathsf{lift}_k(st) \sqsubseteq st(j)$, where st is $st(i)$ without its top element; in particular, the last formula checks that $st(j)$ is one element shorter than $st(i)$ and that the unused part of the stack contains the default value 0.

4 Proof of non-interference

The following theorem relates typability and the fact that the program verifies correctly in the bytecode logic. It says that whenever a program with annotations proposed in Sect. 3 successfully verifies it also successfully typechecks. This property and the main theorem of [4] (see Sect.6) imply the non-interference.

Please recall that like [4], we assume that functions classAnalysis, excAnalysis, nbLocs, nbArgs, Handler are correct.

Theorem 4.1 `(typechecking and verifiability)`
Let P be a Java bytecode program, $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ a desired security policy, and cdr *a control dependence regions structure satisfying SOAP. Let* TR *be the translation defined in Sect. 3 that adds base logic annotations to P. For every security environment family $\{\mathsf{se}_i : PP \to S\}_{i \in S}$ and a family of functions $\{\mathsf{st}_i : PP \to S^*\}_{i \in S}$ such that $st_i(0) = \varepsilon$ for all i,*

$\Rightarrow$ *if the annotated program $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly then P with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and* cdr *is typable,*

$\Leftarrow$ *if the program P with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and* cdr *is typable with* se, st, *and all $Q_i(c0, c)$ in QNI_m in (1) on page 10 are true then the annotated program $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly.*

Proof:
We present here a sketch of the proof only.

($\Rightarrow$) Suppose that the annotated program $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ 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 $\mathsf{Policies}_\Gamma(m)$. We need to verify that the condition in Def. 1 is fulfilled. Let sgn be a signature corresponding to security level s. We take se and st as above. $\mathsf{st}_s(0) = \varepsilon$ 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 $\texttt{inst}(i) = \texttt{ifeq}\ j$. In other cases the proof is similar.

ifeq As the instruction at i is $\texttt{ifeq}\ j_0$ we must only ensure condition (1) of Def. 1. In case of the `ifeq` instruction, the body of the formula $N(i)(s)$ is $(i \mapsto^e j_0 ==> (\mathsf{Reg}^{\texttt{ifeq},s}(j_0)))$. This formula ensures that in case $j = j_0$ or $j = i+1$ the formula $\mathsf{Reg}^{\texttt{ifeq},s}(j_0)$ 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 $\mathsf{Reg}^{\texttt{ifeq},s}(j_0)$ holds, we obtain $\forall j' \in \texttt{region}(i, \emptyset),\ k \leq se(j')$ as k is identified with $gst[s][i][\mathsf{cntr}]$. The second and the third \forall subformula of $\mathsf{Reg}^{\texttt{ifeq},s}(j_0)$ ensure that $\mathsf{lift}_k(st) \sqsubseteq st(j)$ (where $j = i+1$ or j_0).

($\Leftarrow$) Suppose that the program P with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable with se and st. We have to ensure that each $\mathsf{QNI}_m(i)$, for i being a label in the method m, holds. As $Q_i(c0, c)$ is true, it is enough to check that each $N(i)(j)$ holds for $j \in \mathcal{S}$. Each of the $N(i)(s)$ has similar structure presented in (2). It is enough to show that one of the corresponding $\mathsf{Reg}_l^{\texttt{inst}(i),s}(\boldsymbol{p_l})$ holds in case $i \mapsto^e j$ (or in case $i \mapsto^e$). As the method is typable, we know that $\Gamma, \mathsf{ft}, \texttt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies s$ (or $\Gamma, \mathsf{ft}, \texttt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies$) can be inferred. This is done with one of the rules, say l-th. Now, we have to make sure that the corresponding translation formula $\mathsf{Reg}_l^{\texttt{inst}(i),s}(\boldsymbol{p_l})$ holds. We show this in case $\texttt{inst}(i)$ is $\texttt{ifeq}\ j$.

- the first subformula of (3) holds as the typing rule guarantees that the property $\forall j' \in \texttt{region}(i, \emptyset), k \leq \mathsf{se}(j')$ holds,
- the second subformula of (3) holds as the typability requires that $\mathsf{lift}_k(\mathsf{st}) \sqsubseteq \mathsf{st}(j)$, where st is $st(i)$ without its top element,
- the third subformula of (3) holds as the $\mathsf{st}(j)$ 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. □

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 specificational extension of $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ that does not

conflict with $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$. *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 $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ then the values of all the variables used in the translation are the same. In this light, the logical values of the formulae 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 Q_m 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 Ins_m. 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 on the stack 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 *gst* arrays that correspond to i. We can exploit a `set` instruction in $\mathsf{Ins}_m(i)$ to change *gsgn* and *gst* right before the instruction that requires the declassification and revert it back right before the next one. In this way we obtain 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 essentially 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 flavour similar to the approaches [3,6,13]. 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 $\forall^*$ for-

mulae with three predicates $\leq, <$ and $=$. This class is decidable according to the classical result by Bernays, Schönfinkel (see e.g. [9]). Interestingly enough, this class is more powerful and the formulae could take up, roughly, the form of $\exists \mathsf{se}, \mathsf{st} \forall \ldots$ where se is the security environment and st 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 several different `assert` 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 programme. 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 momenta tool to transform the inferences in the information flow type system to BML using the translation is under the development.

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

References

[1] Amtoft, T., S. Bandhakavi and A. Banerjee, *A logic for information flow in object-oriented programs*, SIGPLAN Not. **41** (2006), pp. 91–102, an extended version in the report KSU CIS-TR-2005-1.

[2] Appel, A. W. and A. P. Felty, *A semantic model of types and machine instructions for proof-carrying code*, in: *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000), pp. 243–253.

[3] Barthe, G., P. R. D'Argenio and T. Rezk, *Secure information flow by self-composition*, in: *CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations* (2004), p. 100.

[4] Barthe, G., D. Pichardie and T. Rezk, *A certified lightweight non-interference Java bytecode verifier*, Technical report, INRIA (2006).

[5] Barthe, G., D. Pichardie and T. Rezk, *A Certified Lightweight Non-Interference Java Bytecode Verifier*, in: *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science **4421** (2007), pp. 125–140.

[6] Benton, N., *Simple relational correctness proofs for static analyses and program transformations*, in: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2004), pp. 14–25.

[7] Beringer, L. and M. Hofmann, *A bytecode logic for JML and types*, in: *Asian Programming Languages and Systems Symposium*, LNCS (2006), pp. 389–405.
URL http://www.tcs.informatik.uni-muenchen.de/~beringer/

[8] Beringer, L. and M. Hofmann, *Secure information flow and program logics*, in: *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium* (2007), pp. 233–248.

[9] Börger, E., E. Grädel and Y. Gurevich, "The classical decision problem," Springer-Verlag, 1997.

[10] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino and E. Poll, *An overview of JML tools and applications*, in: T. Arts and W. Fokkink, editors, *Workshop on Formal Methods for Industrial Critical Systems*, ENTCS **80** (2003), pp. 73–89.

[11] Burdy, L., M. Huisman and M. Pavlova, *Preliminary design of BML: A behavioral interface specification language for Java bytecode*, in: *Fundamental Approaches to Software Engineering (FASE 2007)*, Lecture Notes in Computer Science **4422** (2007), pp. 215–229.

[12] Consortium, M., *Deliverable 3.1: Bytecode specification language and program logic* (2006), available online from http://mobius.inria.fr.

[13] Hahnle, R., J. Pan, P. Rummer and D. Walter, *Integration of a security type system into a program logic*, Theoretical Computer Science **402** (2008), pp. 172–189.

[14] Hennessy, M., *The security pi-calculus and non-interference*, Journal of Logic and Algebraic Programming **63** (2004), pp. 3–34.

[15] Honda, K. and N. Yoshida, *Noninterference through flow analysis*, Journal of Functional Programming **15** (2005), pp. 293–349.

[16] Hristova, K., T. Rothamel, Y. A. Liu and S. D. Stoller, *Efficient type inference for secure information flow*, in: *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security* (2006), pp. 85–94.

[17] Hunt, S. and D. Sands, *On flow-sensitive security types*, in: *Principles of Programming Languages* (2006).
URL http://mobius.inria.fr/twiki/pub/Publications/WebHome/Hunt-Sands-POPL06.pdf

[18] Jacobs, B. and E. Poll, *A logic for the Java Modeling Language JML*, in: H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, LNCS **2029** (2001), pp. 284–299.

[19] Leavens, G., A. Baker and C. Ruby, *Preliminary design of JML: A behavioral interface specification language for Java*, Technical Report TR 98-06y, Iowa State University (1998), (revised since then 2004).

[20] Leavens, G., E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok and J. Kiniry, "JML Reference Manual," (2005), in Progress. Department of Computer Science, Iowa State Univer sity. Available from http://www.jmlspecs.org.

[21] Sabelfeld, A. and A. Myers, *Language-Based Information-Flow Security*, IEEE Journal on Selected Areas in Communication **21** (2003), pp. 5–19.

Experiments with Non-Termination Analysis for Java Bytecode

Étienne Payet[1]

IREMIA
Université de La Réunion
France

Fausto Spoto[2]

Dipartimento di Informatica
Università di Verona
Italy

Abstract

Non-termination analysis proves that programs, or parts of a program, do not terminate. This is important since non-termination is often an unexpected behaviour of computer programs and exposes a bug in their code. While research has found ways of proving non-termination of logic programs and of term rewriting systems, this is hardly the case for imperative programs. In this paper, we describe and experiment with a technique for proving non-termination of imperative, bytecode programs by relating their non-termination to that of a (constraint) logic program. Moreover, we show that our non-termination test effectively helps a termination test, by avoiding expensive search for termination proofs of those portions of the code where such proofs do not exist.

Keywords: Java, Java bytecode, static analysis, termination, non-termination

1 Introduction

Java bytecode [8] is the result of the compilation of Java, as well as of other programming languages. It is a low-level, object-oriented, type-safe language which is distributed in a machine-independent format, hence executable on different architectures. It is the target of choice for the compilation of applications that must be downloaded from the net into client computers or mobile phones. The recent Android system by Google [1] uses the Java bytecode as the target of the compilation of Android programs, before translating it into a machine-centered lower-level bytecode.

[1] Email: epayet@univ-reunion.fr
[2] Email: fausto.spoto@univr.it

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

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 overuse 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 violation 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 *Julia+BinTerm* 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 results from non-termination analysis of logic programs [10]; 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

not terminate because of the approximations induced by the abstraction from P to P_{CLP}.

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 [3], 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 P_{CLP} over linear constraints, whose predicates $b(vars)$ correspond to each basic block b of P and $vars$ 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 [19], *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 $b(vars)$ has only terminating computations in P_{CLP}, for all fixed integer values for $vars$, 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 $b(vars)$ does not terminate for some fixed initial integer values for $vars$, although all executions of P starting at b do terminate. This is due to the approximations done during 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 $p(\tilde{X}) \leftarrow c, q(\tilde{Y})$, 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 optimisations 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*, in the sense 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 ($CLP(\mathbb{PL})$).

We let $\tilde{t}$ denote a sequence of terms, $\tilde{X}$ and $\tilde{Y}$ denote sequences of distinct variables, p and q denote predicate symbols and c denote a path-length constraint. An *atom* has the form $p(\tilde{t})$ where the length of $\tilde{t}$ equals the arity of p. A *query* has the form $\langle p(\tilde{X}) \,|\, c\rangle$. A *clause* has the form $p(\tilde{X}) \leftarrow c, q(\tilde{Y})$ where $\tilde{X}$ and $\tilde{Y}$ are disjoint and the variables occurring in c necessarily occur in $\tilde{X} \cup \tilde{Y}$. A $CLP(\mathbb{PL})$ *program* is a finite set of clauses. We use $\exists_{\tilde{X}} c$ as a shortcut for $\exists X_1 \ldots \exists X_n c$ where $X_1, \ldots, X_n := \tilde{X}$. The *projection* of c onto the sequence $\tilde{X}$ is denoted by $\overline{\exists}_{\tilde{X}} c$ and is the constraint $\exists_{Var(c)\setminus\tilde{X}} c$, where $Var(c)$ is the set of variables occurring in c. The *set described by a query* $Q := \langle p(\tilde{X}) \,|\, c\rangle$ is denoted by $Set(Q)$; it consists of all the atoms of the form $p(v(X_1), \ldots, v(X_n))$ where $X_1, \ldots, X_n := \tilde{X}$ and v is a ground solution of c. We say that $Set(Q)$ is *non-terminating wrt.* a $CLP(\mathbb{PL})$ program P when for all $p(v(X_1), \ldots, v(X_n)) \in Set(Q)$, the query

$$\langle p(X_1, \ldots, X_n) \,|\, X_1 = v(X_1), \ldots, X_n = v(X_n)\rangle$$

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 P, 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 ([10]) *Let $p(\tilde{X}) \leftarrow c, p(\tilde{Y})$ be a recursive clause in a $CLP(\mathbb{PL})$ program P. If $Set(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq Set(\langle p(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ then $Set(\langle p(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ is non-terminating wrt. P.* □

Theorem 3.1 means that if the set of values assigned to $\tilde{Y}$ by all the solutions of c is included in the set of values assigned to $\tilde{X}$ by all the solutions of c, then any value assigned to $\tilde{X}$ by a solution of c provides a non-terminating ground query. Indeed, intuitively, the constraint $\overline{\exists}_{\tilde{X}} c$ is the guard of the clause and $Set(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq Set(\langle p(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ means that every output value of the clause satisfies this guard. Hence, if a value satisfies the guard, then it enters the clause and the corresponding output satisfies the guard, so this output can also enter the clause and the next output satisfies the guard, and so on. Notice that the converse of the implication in Theorem 3.1 does not always hold: consider for instance the recursive clause $p(X) \leftarrow X \geq 3, p(Y)$; we have that $Set(\langle p(X) \,|\, \overline{\exists}_X X \geq 3\rangle)$, *i.e.* $Set(\langle p(X) \,|\, X \geq 3\rangle)$, is non-terminating wrt. this clause although $Set(\langle p(Y) \,|\, \overline{\exists}_Y X \geq 3\rangle)$, *i.e.* $Set(\langle p(Y) \,|\, true\rangle)$, is not included in $Set(\langle p(X) \,|\, X \geq 3\rangle)$.

Theorem 3.2 ([10]) *Let $q(\tilde{X}) \leftarrow c, p(\tilde{Y})$ be a clause in a $CLP(\mathbb{PL})$ program P and Q be a query such that $Set(Q)$ is non-terminating wrt. P. If $Set(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq Set(Q)$ then $Set(\langle q(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ is non-terminating wrt. P.* □

The intuition of Theorem 3.2 is that any value $q(\tilde{x})$ in $Set(\langle q(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ satisfies $\overline{\exists}_{\tilde{X}} c$, the guard of the clause, and the corresponding output $p(\tilde{y})$ is included in $Set(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle)$. As $Set(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq Set(Q)$ and $Set(Q)$ is non-terminating *wrt.* P, then $p(\tilde{y})$ does not terminate *wrt.* P, so $q(\tilde{x})$ 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

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 $ins^{\mathbb{PL}}$ be a correct approximation of its behaviour, *i.e.*, a constraint over input variables $\check{v}$ and output variables $\hat{v}$. This approximation is *exact* if and only if, for all input states $\check{\sigma}$ and output variable $\hat{\sigma}$ satisfying the static information at *ins* (number and type of local variables and stack elements), whenever $\{\check{v} \mapsto \mathit{pathlength}(\check{\sigma}(v))\} \cup \{\hat{v} \mapsto \mathit{pathlength}(\hat{\sigma}(v))\} \models ins^{\mathbb{PL}}$ then $\sigma(\check{\sigma}) = \hat{\sigma}$. □

Consider for instance the program *Add1*:

```
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

$$add_q^{\mathbb{PL}} = \mathit{Unchanged}_q(\#l, \#s - 2) \cup \{\check{s}^{\#s-2} + \check{s}^{\#s-1} = \hat{s}^{\#s-2}\}$$

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 v at the beginning of the execution of the bytecode, written as $\check{v}$, and variables at its end, written as $\hat{v}$. 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 ($\hat{s}^{\#s-2}$) holds a value which is equal the addition of the former two topmost stack elements ($\check{s}^{\#s-2}$ and $\check{s}^{\#s-1}$). This approximation is exact since, for every couple of input state $\check{\sigma}$ and output state $\hat{\sigma}$ satisfying the static information at this

bytecode and the approximation above, we must have that the local variables have the same values in $\check{\sigma}$ 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 $\check{\sigma}$, so that those states are such that $add_q(\check{\sigma}) = \hat{\sigma}$. The corresponding $CLP(\mathbb{PL})$ program $Add1_{CLP}$ is:

$$entry(IL2) \leftarrow \{IL2 - OL2 = -1,\ -IL2 \geq -4,\ IL2 \geq 2\}, entry(OL2)$$

The predicate *entry* denotes the entry point of the loop of the program; local variable 2 implements `i` 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 $IL2 - OL2 = -1$ (that is, local variable 2, which is `i`, 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 *Add1* terminates also. If we turn *Add1* into the non-terminating program:

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

we get the $CLP(\mathbb{PL})$ program $Add2_{CLP}$:

$$entry(IL2) \leftarrow \{IL2 - OL2 = 1,\ -IL2 \geq -2\}, entry(OL2)$$

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

$$Set(\langle entry(OL2) \,|\, -OL2 \geq -1\rangle) \subseteq Set(\langle entry(IL2) \,|\, -IL2 \geq -2\rangle)\ .$$

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:

```
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 $CLP(\mathbb{PL})$ program $Add3_{CLP}$:

$$entry(IL3) \leftarrow \{OL3 = 2\}, block(OL3)$$

$$block(IL3) \leftarrow \{IL3 - OL3 = 1,\ -IL3 \geq -2\}, block(OL3)$$

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

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 *wrt.* $Add3_{CLP}$ where

$$Q := \langle block(IL3) \,|\, -IL3 \geq -2 \rangle \ .$$

Notice that we have to infer a non-terminating query of the form $\langle entry(\cdots) \,|\, \cdots \rangle$ 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

$$Set(\langle block(OL3) \,|\, OL3 = 2 \rangle) \subseteq Set(Q) \ .$$

Hence, by Theorem 3.2 applied to the first clause of $Add3_{CLP}$ and to Q, we have that $Set(\langle entry(IL3) \,|\, true \rangle)$ is non-terminating *wrt.* $Add3_{CLP}$ (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 *Add2* into a separated method, such as in:

```
public class Add4 {
    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 $CLP(\mathbb{PL})$ program $Add4_{CLP}$:

$$entry(IL1) \leftarrow \{IL1 - OL1 = 1,\ -IL1 \geq -2\}, entry(OL1)$$

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:

```
public class Rec1 {
    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);
    }
}
```

The $CLP(\mathbb{PL})$ program $Rec1_{CLP}$:

$$entry(IL0) \leftarrow \{IL0 - OL0 = 1,\ IL0 \geq 1,\ -IL0 \geq -2\}, entry(OL0)$$

terminates, hence by Theorem 2.1 we conclude that *Rec1* terminates. If we turn *Rec1* into the following non-terminating program (where the programmer forgot the base case in the recursive method):

```
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 $CLP(\mathbb{PL})$ program $Rec2_{CLP}$:

$$entry(IL0) \leftarrow \{IL0 - OL0 = 1,\ -IL0 \geq -2\}, entry(OL0)$$

By Theorem 3.1, $Rec2_{CLP}$ does not terminate. As the approximation of the bytecode program corresponding to *Rec2* is exact, we can safely conclude that *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:

```
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 *iter* (intended to perform an iteration over a list) contains a bug since it recurs on the same list rather than on its tail (`iter()` instead of `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 *main* is not cyclical. The corresponding $CLP(\mathbb{PL})$ program $List_{CLP}$:

$$entry \leftarrow true,\ entry$$

(*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

$$mul_q^{\mathbb{PL}} = Unchanged_q(\#l, \#s - 2)$$

($\#l$ 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:

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

terminates. Notice that the guard of the loop involves a multiplication. The corresponding $CLP(\mathbb{PL})$ program Mul_{CLP}:

$$entry(IL2) \leftarrow \{IL2 - OL2 = -1,\ IL2 \geq 2\}, entry(OL2)$$

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

$$Set(\langle entry(OL2) \,|\, OL2 \geq 3\rangle) \subseteq Set(\langle entry(IL2) \,|\, IL2 \geq 2\rangle)\ .$$

Therefore, by Theorem 3.1, the non-empty set $Set(\langle entry(IL2) \,|\, IL2 \geq 2\rangle)$ is non-terminating *wrt.* Mul_{CLP}. However, the non-termination of *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 f* instruction takes the reference to an object o located on top of the stack and replaces it with the value of $o.f$. In [19] we defined

$$getfield_q^{\mathbb{PL}}\ f = Unchanged_q(\#l, \#s - 1)$$

whenever 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:

```
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. The corresponding $CLP(\mathbb{PL})$ program $Field_{CLP}$:

$$entry(IL2) \leftarrow \{OL2 - IL2 = 1\}, entry(OL2)$$

does not terminate as the projection of the constraint of its clause onto $IL2$ or onto $OL2$ is the always satisfiable constraint $true$ and we have

$$Set(\langle entry(OL2) \mid true \rangle) \subseteq Set(\langle entry(IL2) \mid true \rangle) \ .$$

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.

P	number of methods in P	number of clauses in P_{CLP}
JavaCup	270	170
JLex	137	356
Kitten	2149	1224

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.

P	clauses removed	non-termination analysis	*BinTerm* on P'_{CLP}	*BinTerm* on P_{CLP}
JavaCup	113	0.09s	3.90s	5.66s
JLex	204	0.20s	21.20s	55.30s
Kitten	288	0.68s	99.52s	100.99s

In these experiments on large programs, the computational overhead of the non-

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* translation of the original bytecode program.

References

[1] *Android - An Open Handset Alliance Project*, `http://code.google.com/android/`.

[2] Bol, R. N., K. R. Apt and J. W. Klop, *An Analysis of Loop Checking Mechanisms for Logic Programs*, Theoretical Computer Science **86** (1991), pp. 35–79.

[3] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in: *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, 1977, pp. 238–252.

[4] De Schreye, D., K. Verschaetse and M. Bruynooghe, *A Practical Technique for Detecting non-Terminating Queries for a Restricted Class of Horn Clauses, using Directed, Weighted Graphs*, in: *Proc. of ICLP'90* (1990), pp. 649–663.

[5] Giesl, J., R. Thiemann and P. Schneider-Kamp, *Proving and Disproving Termination of Higher-order Functions*, in: B. Gramlich, editor, *Proc. of the 5th International Workshop on Frontiers of Combining Systems (FroCoS'05)*, Lecture Notes in Artificial Intelligence **3717** (2005), pp. 216–231.

[6] Godefroid, P., N. Klarlund and K. Sen, *DART: Directed Automated Random Testing*, in: V. Sarkar and M. W. Hall, editors, *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)* (2005), pp. 213–223.

[7] Gupta, A., T. Henzinger, R. Majumdar, A. Rybalchenko and R. Xu, *Proving non-Termination*, in: G. Necula and P. Wadler, editors, *Proc. of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)* (2008), pp. 147–158.

[8] Lindholm, T. and F. Yellin, "The Java^{TM} Virtual Machine Specification," Addison-Wesley, 1999, second edition.

[9] Payet, E., *Loop Detection in Term Rewriting Using the Eliminating Unfoldings*, Theoretical Computer Science **403** (2008), pp. 307–327.

[10] Payet, E. and F. Mesnard, *A non-Termination Criterion for Binary Constraint Logic Programs*, Technical report, IREMIA, Université de La Réunion (2008), available at `http://arxiv.org/abs/0807.3451`.

[11] Payet, E. and F. Mesnard, *Non-Termination Inference of Logic Programs*, ACM Transactions on Programming Languages and Systems **28, Issue 2** (March 2006), pp. 256–289.

[12] Rossignoli, S. and F. Spoto, *Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions*, in: E. A. Emerson and K. S. Namjoshi, editors, *Proc. of the 7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06)*, Lecture Notes in Computer Science **3855** (2006), pp. 95–110.

[13] Secci, S. and F. Spoto, *Pair-Sharing Analysis of Object-Oriented Programs*, in: C. Hankin and I. Siveroni, editors, *Proc. of Static Analysis Symposium (SAS'05)*, Lecture Notes in Computer Science **3672**, London, UK, 2005, pp. 320–335.

[14] Sen, K., D. Marinov and G. Agha, *CUTE: a Concolic Unit Testing Engine for C*, in: M. Wermelinger and H. Gall, editors, *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2005), pp. 263–272.

[15] Shen, Y.-D., J.-H. You, L.-Y. Yuan, S. Shen and Q. Yang, *A Dynamic Approach to Characterizing Termination of General Logic Programs*, ACM Transactions on Computational Logic **4** (2003), pp. 417–434.

[16] Shen, Y.-D., L.-Y. Yuan and J.-H. You, *Loops Checks for Logic Programs with Functions*, Theoretical Computer Science **266** (2001), pp. 441–461.

[17] Spoto, F., P. M. Hill and E. Payet, *Path-Length Analysis for Object-Oriented Programs*, in: *First International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*, Vienna, Austria, 2006, available at the web address `http://profs.sci.univr.it/~spoto/papers.html`.

[18] Spoto, F., L. Lu and F. Mesnard, *Using CLP Simplifications to Improve Java Bytecode Termination Analysis*, submitted for publication to Bytecode'09.

[19] Spoto, F., F. Mesnard and E. Payet, *A Termination Analyser for Java Bytecode Based on Path-Length*, submitted for publication in August 2007.

[20] Velroyen, H. and P. Rümmer, *Non-Termination Checking for Imperative Programs*, in: B. Beckert and R. Hähnle, editors, *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Lecture Notes in Computer Science **4966** (2008), pp. 154–170.

[21] Waldmann, J., *Matchbox: A Tool for Match-bounded String Rewriting*, in: V. van Oostrom, editor, *Proc. of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, Lecture Notes in Computer Science **3091** (2004), pp. 85–94.

[22] Waldmann, J., *Compressed Loops (draft)* (2007), available at `http://dfa.imn.htwk-leipzig.de/matchbox/methods/`.

[23] Zankl, H. and A. Middeldorp, *Nontermination of String Rewriting using SAT*, in: *Proc. of the 9th International Workshop on Termination (WST'07)*, 2007, pp. 52–55.

[24] Zantema, H., *Termination of String Rewriting Proved Automatically*, Journal of Automated Reasoning **34** (2005), pp. 105–139.

Jalapa: Securing Java with Local Policies

Massimo Bartoletti

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy

Gabriele Costa

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

Roberto Zunino

Dipartimento di Ingegneria e Scienza dell'Informazione, Università di 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 sytem, 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

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

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 [1,7,11]. 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 implementated 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.

2 Securing Java with local usage policies

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`.

```
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 `run()`. 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.

```
name: plugin-out
aliases:
read  := FileInputStream.<init>()
read  := Browser.readCookie(URL u)
write := Socket.getOutputStream()
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- read --> q1
q1 -- write --> fail
```

```
name: plugin-cookies
aliases:
cookie(u) := Browser.writeCookie(URL u,String m)
cookie(u) := Browser.readCookie(URL u)
init(p,u) := (p:Plugin).<init>(URL u)
start(p)  := (p:Plugin).doIt()
states: q0 q1 q2 fail
start: q0
final: fail
trans:
q0 -- init(p,u) --> q1
q1 -- start(p) --> q2
q2 -- cookie(u') --> fail when u'!=u
q2 -- start(p') --> q1 when p'!=p
```

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

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` preludes to the transition relation of the automaton. In our example, a transition from `q0` to `q1` occurs upon reading any file or cookie. A transition from `q1` 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 `q0`. 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 $\mathcal{M}$ 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 $\texttt{W}_\texttt{C}$ for the class `C` declares exactly the same methods of `C`, implements all the interfaces of `C`, and extends the superclass of `C`. Indeed, $\texttt{W}_\texttt{C}$ can replace `C` in any context, in that it admits the same operations of `C`. A method `m` of $\texttt{W}_\texttt{C}$ can be either monitored or not. If the corresponding method `m` of `C` does not belong to $\mathcal{M}$, then $\texttt{W}_\texttt{C}$`.m` simply calls `C.m`. Otherwise, $\texttt{W}_\texttt{C}$`.m` 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, i.e. a library that contains the resources necessary to the monitoring process. Note that our instrumentation produces a stand-alone application, requiring no custom JVM and no further external components.

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 $((\mathtt{O}_1, \ldots, \mathtt{O}_k), \mathtt{Q})$, where $(\mathtt{O}_1, \ldots, \mathtt{O}_k)$ 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 available 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 implementation 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 := C1.m1(...) { C2.m2(...) }`, meaning that the event `ev` is fired whenever the method `m1` of class `C1` is invoked within the scope of the method `m2` of class `C2`. 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 bytecode, 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

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

[1] Abadi, M. and C. Fournet, *Access control based on execution history*, in: *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.

[2] Bartoletti, M., *Usage automata* (2009), to appear in ARSPA-WITS.

[3] Bartoletti, M., P. Degano, G. L. Ferrari and R. Zunino, *Types and effects for resource usage analysis*, in: *Proc. Fossacs*, 2007.

[4] Bartoletti, M., P. Degano, G. L. Ferrari and R. Zunino, *Model checking usage policies*, in: *Proc. Trustworthy Global Computing*, 2008.

[5] Bartoletti, M., P. Degano, G. L. Ferrari and R. Zunino, *Semantics-based design for secure web services*, IEEE Transactions on Software Engineering **34** (2008).

[6] Bartoletti, M. and R. Zunino, *LocUsT: a tool for checking usage policies*, Technical Report TR-08-07, Dip. Informatica, Univ. Pisa (2008).

[7] Edjlali, G., A. Acharya and V. Chaudhary, *History-based access control for mobile code*, in: *Secure Internet Programming*, Lecture Notes in Computer Science **1603**, 1999.

[8] Fournet, C. and A. D. Gordon, *Stack inspection: theory and variants*, ACM Transactions on Programming Languages and Systems **25** (2003), pp. 360–399.

[9] Gong, L., "Inside Java 2 platform security: architecture, API design, and implementation," Addison-Wesley, 1999.

[10] *Jalapa: Securing Java with Local Policies*, `http://jalapa.sourceforge.net`.

[11] Skalka, C. and S. Smith, *History effects and verification*, in: *Proc. APLAS*, 2004.

[12] Welch, I. and R. J. Stroud, *Kava - using byte code rewriting to add behavioural reflection to Java*, in: *USENIX Conference on Object-Oriented Technology*, 2001.