Modular Termination Analysis of Java Bytecode and its Application to phoneME Core Libraries

D. Ramírez-Deantes[1], J. Correas[2], and G. Puebla[1]

[1] DLSIIS, Technical University of Madrid (UPM), Spain
[2] DSIC, Complutense University of Madrid (UCM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming and, recently, also for imperative and Object Oriented (OO) languages. In fact, there exist termination analyzers for OO which are capable of proving termination of medium size applications by means of *global* analysis, in the sense that all the code used by such applications has to be proved terminating. However, global analysis has important weaknesses, such as its high memory requirements and its lack of efficiency, since often some parts of the code have to be analyzed over and over again, libraries being a paramount example of this. In this work we present how to extend the termination analysis in the COSTA system in order to make it modular by allowing separate analysis of individual methods. The proposed approach has been implemented. We report on its application to the termination analysis of the core libraries of the phoneME project, a well-known open source implementation of Java Micro Edition (JavaME), a realistic but reduced version of Java to be run on mobile phones and PDAs. We argue that such experiments are relevant, since handling libraries is known to be one of the most relevant open problems in analysis and verification of real-life applications. Our experimental results show that our proposal dramatically reduces the amount of code which needs to be handled in each analysis and that this allows proving termination of a good number of methods for which global analysis is unfeasible.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination. In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [16], and logic and functional languages [18, 9, 17].

Termination-proving techniques are also emerging in the imperative paradigm [7, 10, 16] and the object oriented (OO for short) paradigm, where static analysis tools such as Julia [25], AProVE [21], and COSTA [1] are able to prove termination of non-trivial medium-size programs. In the context of OO languages, we focus on the problem of proving whether the execution of a method m terminates for any possible input value which satisfies m's precondition, if any. Solving this problem requires, at least in principle, a *global analysis*, since proving that the execution of m terminates requires proving termination of all methods transitively invoked during m's execution. In fact, the three analysis tools for OO code mentioned above require the code of all methods reachable from m to be available to the analyzer and aim at proving termination of all the code involved. Though this approach is valid for medium-size programs, we quickly get into scalability problems when trying to analyze larger programs. It is thus required to reach some degree of *compositionality* which allows decomposing the analysis of large programs into the analysis of smaller parts.

In this work we propose an approach to the termination analysis of large OO programs which is compositional and we (mostly) apply it by analyzing a method at a time. We refer to the latter as *modular*, i.e., which allows reasoning on a method at a time. Our approach provides several advantages: first, it allows the analysis of larger programs, since the analyzer does not need to have the complete code of the program nor the intermediate results of the analysis in memory. Second, methods are often used by several other methods. The analysis results of a shared method can be reused for multiple uses of the method.

The approach presented is flexible in the level of granularity: it can be used in a component-based system at the level of components. A specification can be generated for a component C by analyzing its code, and it can be deployed together with the component and used afterwards for analyzing other components that depend on this one. When analyzing a component-based application that uses C, the code of C does not need to be available at analysis time, since the specification generated can be used instead.

In order to evaluate the effectiveness of our approach, we have extended the COSTA analyzer to be able to perform modular termination analysis and we have applied the improved system to the analysis of the phoneME implementation of the core libraries of JavaME. Note that analysis of API libraries is quite challenging and a significant stress test for the analyzer for a number of reasons which are discussed in more detail in Section 5 below. The main contribution of this paper is that it provides a practical framework for the modular analysis of Java bytecode, illustrating its applicability to real programs by analyzing phoneME librares. These contributions are detailed from Section 4 onwards.

2 Non-Modular Termination Analysis in COSTA

COSTA (see [4] and its references) is a cost [2] and termination [1] analyzer for Java bytecode. COSTA receives as input the signature of the method m whose termination (or cost) we want to infer. Method m is assumed to be available

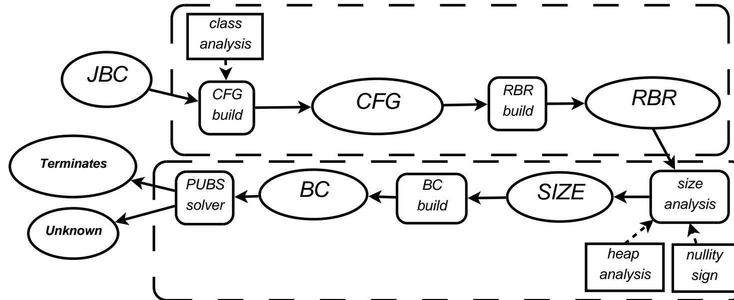

Fig. 1. Architecture of COSTA

in the classpath or default Java run-time environment (jre for short) libraries, together with all other methods and classes transitively invoked by m. Since there can be many more classes and methods in the classpath and jre than those reachable from m, a first step during analysis consists in identifying a set M of methods which includes all methods reachable from m. This phase is sometimes referred to as *program extraction* or *application extraction*. Then, COSTA performs a *global analysis* since, not only m, but all methods in the program M are analyzed.

We now briefly describe the overall architecture of COSTA, which is graphically represented in Figure 1. More details can be found in [3]. The dashed frames represent the two main phases of the analysis: (i) consists of extracting a program M from the method m plus the transformation of the bytecode for all methods in M into a suitable internal representation; and (ii) the actual static analysis. Input and output of the system are depicted on the left: by *JBC* we denote the bytecode of all classes in the classpath and jre plus the signature of a method and yields information about termination, indicated by *Terminates* (the analyzer has proved that the program terminates for all valid inputs) or *Unknown* (otherwise). Ellipses (e.g. *CFG*) represent *what* the system produces at each intermediate stage of the analysis; rounded boxes (e.g. "*CFG build*") indicate the *main steps* of the analysis process; square boxes (e.g. *class analysis*), which are connected to the main steps by dashed arrows, denote auxiliary analyses which allow obtaining more precise results. During the first phase, depicted in the upper half of the figure, the incoming *JBC* is transformed into a *rule-based representation* (*RBR*). In the second phase, depicted in the lower half of the figure, the system performs the actual termination analysis on the RBR.

2.1 From the Bytecode to the Rule-based Representation

Generation of Control Flow Graphs guided by Class Analysis COSTA transforms the bytecode of a method into *Control Flow Graphs* (CFGs) by using techniques from compiler theory. As regards *Virtual invocation*, computing a precise approximation of the methods which can be executed at a given program

point is not trivial. As customary in the analysis of OO languages, COSTA uses *class analysis* [24] (or points-to analysis) in order to precisely approximate this information. First, the CFG of the initial method is built, and class analysis is applied in order to approximate the possible runtime classes at each program point. This information is used to *resolve* virtual invocations. Methods which can be called at runtime are loaded, and their corresponding CFGs are constructed. Class analysis is applied to their body to include possibly more classes, and the process continues iteratively. Once a fixpoint is reached, it is guaranteed that all reachable methods have been loaded, and the corresponding CFGs have been generated.

As regards *exceptions*, COSTA handles internal exceptions (i.e., those associated to bytecodes as stated in the JVM specification), exceptions which are thrown (bytecode athrow) and possibly propagated back in methods, as well as finally clauses. Exceptions are handled by adding edges to the corresponding handlers. COSTA provides the options of ignoring only internal exceptions, all possible exceptions or considering them all.

Rule-based Representation Given a method m and its CFGs, a RBR for m is obtained by producing, for each basic block m_j in its CFGs, a rule which (1) contains the set of bytecode instructions within the basic block; (2) if there is a method invocation within the instructions, includes a call to the corresponding rule; and (3) at the end, contains a call to a *continuation rule* m_j^c which includes mutually exclusive rules to cover all possible continuations from the block. Note that several rules may be produced with the same name. A *procedure* P is the set of all rules with name P.

2.2 Context-Sensitive (Pre-)Analyses to Improve Accuracy

COSTA performs three context-sensitive analyses on the RBR based on abstract interpretation [12]: *nullity*, *sign* and *heap* analysis. These analyses improve the accuracy (and efficiency) of subsequent steps inferring information from individual bytecodes, and propagating it via a standard, top-down *fixpoint* computation.

Nullity Analysis aims at keeping track of reference variables which are definitely *null* or are definitely *non-null*. For instance, the bytecode $\mathsf{new}(s_i)$ allows assigning the abstract value *non-null* to s_i. The results of nullity analysis often allow removing rules corresponding to NullPointerException.

Sign Analysis aims at keeping track of the sign of variables. The abstract domain contains the elements $\geq$, $\leq$, $>$, $<$, $=0$, $\neq 0$, $\top$ and $\bot$, partially ordered in a lattice. For instance, sign analysis of $\mathsf{const}(s_i, V)$ evaluates the integer value V and assigns the corresponding abstract value $=0$, $>$ or $<$ to s_i, depending, resp., on if V is zero, positive or negative [12]. Knowing the sign of data allows removing RBR rules for arithmetic exceptions which are never thrown.

Heap Analysis obtains information related to variables and arguments located in the heap, a global data structure which contains objects (and arrays) allocated by the program. Infers properties like *constancy* and *cyclicity* of variables and arguments, and *sharing*, *reachability* and *aliasing* between variables

and arguments in the heap [15]. They are used for inferring sound size relations on objects.

2.3 Size Analysis of Java Bytecode

From the RBR, *size* analysis takes care of inferring the relations between the values of variables at different points in the execution. To this end, the notion of *size measure* is crucial. The size of a piece of data at a given program point is an abstraction of the information it contains, which may be fundamental to prove termination. The COSTA system uses several size measures:

- *Integer-value* maps an integer value to its value (i.e., the size of an integer is the value itself). It is typically used in loops with an integer counter.
- *Path-length* [23] maps an object to the length of the maximum path reachable from it by dereferencing. This measure can be used to predict the behavior of loops which traverse linked data structures, such as lists and trees.
- *Array-length* maps an array to its length and is used to predict the behavior of loops which traverse arrays.

Size analysis works in two phases. In the first one, called *abstract compilation*, each bytecode, call or guard is *abstracted* by *linear constraints* on the size of its variables: for example, $\mathsf{iadd}(s_0, s_1, s_0')$ will be abstracted by the constraint $s_0'{=}s_1{+}s_0$, meaning that the size of s_0 after executing the instruction is the sum of the size of s_0 and s_1 before.

In the second phase, linear constraints replacing parts of the program can be propagated via a standard, bottom-up *fixpoint* computation, in order to combine the information about single rules. The goal of this global analysis is to have *size relations* on variables between the input of a rule (i.e., a block in the CFG) and another one which can be (directly or indirectly) called by the first one.

2.4 Inferring Termination

From the RBR and the results of size analysis, a set of *binary clauses* (*BC* in Figure 1) is produced, which capture calls among blocks together with information on how the values of variables change from one call to another. On such binary clauses, standard termination analysis techniques developed for i.e., termination of logic program can be applied. In particular, COSTA proves termination by using semantic-based techniques, relying on *binary unfolding* combined with *ranking functions*, as those in [9]. This is performed by means of the *PUBS* solver. More details on how termination proofs are performed in COSTA can be found in [1].

3 Abstract Interpretation Fundamentals

Before describing the modular analysis framework, a brief description to abstract interpretation is in order. *Abstract interpretation* [12] is a technique for static program analysis in which execution of the program is simulated on a

description (or abstract) domain (D) which is simpler than the actual (or concrete) domain (C). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle\alpha,\gamma\rangle$: *abstraction* $\alpha : 2^C \rightarrow D$ and *concretization* $\gamma : D\rightarrow 2^C$ which form a Galois connection, i.e.

$$\forall x \in 2^C : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall \lambda \in D : \ \alpha(\gamma(\lambda)) = \lambda.$$

The set of all possible descriptions represents a description domain D which is usually a complete lattice for which all ascending chains are finite. Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^C in some precise sense that depends on the particular abstract domain. A description $\lambda \in D$ *approximates* a set of concrete values $x \in 2^C$ if $\alpha(x) \sqsubseteq \lambda$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during the execution of the program.

In COSTA, abstract interpretation is performed on the rule based representation introduced in Section 2. We first introduce some notation. CP and AP stand for descriptions in the abstract domain. The expression $P{:}CP$ denotes a *call pattern*. This consists of a procedure P together with an entry pattern for that procedure. Similarly, $P \mapsto AP$ denotes an answer pattern, though it will be referred to as AP when it is associated to a call pattern $P{:}CP$ for the same procedure. Since a method is represented in the RBR as a set of interconnected procedures that start from a single particular procedure, the same notation will be used for methods: $m{:}CP$ denotes a call pattern that corresponds to an invocation to method m (i.e., the entry procedure for method m), and $m \mapsto AP$ denotes the answer pattern obtained after analyzing method m.

Context-sensitive abstract interpretation takes as input a program R and an initial call pattern $P{:}CP$, where P is a procedure and CP is a restriction of the values of arguments of P expressed as a description in the abstract domain D and computes a set of triples, denoted $analysis(R, P{:}CP) = \{P_1{:}CP_1 \mapsto AP_1, \ldots, P_n{:}CP_n \mapsto AP_n\}$. In each element $P_i{:}CP_i \mapsto AP_i$, P_i is a procedure and CP_i and AP_i are, respectively, the abstract call and answer patterns.

An analysis is said to be *polyvariant* if more than one triple $P{:}CP_1 \mapsto AP_1, \ldots, P{:}CP_n \mapsto AP_n$ $n \geq 0$ with $CP_i \neq CP_j$ for some i, j may be computed for the same procedure P, while a *monovariant* analysis computes (at most) a single triple $P{:}CP \mapsto AP$ for each procedure (with a call pattern CP general enough to cover all possible patterns that appear during the analysis of the program for P). Although in general context-sensitive, polyvariant analysis algorithms are more precise than those obtained with context-insensitive or monovariant analyses, monovariant algorithms are simpler and have smaller memory requirements. Context-insensitive analysis does not consider call pattern information, and therefore obtains as result of the analysis a set of pairs $\{P_1 \mapsto AP_1, \ldots, P_n \mapsto AP_n\}$, valid for any call pattern.

COSTA includes several abstract interpretation based analyses: nullity and sign are context-sensitive and monovariant, size is context-insensitive, and heap properties analysis [15] is context-sensitive and polyvariant.

4 Extending COSTA to Modular Termination Analysis

As described in Section 2, the termination analysis performed by COSTA is in fact a combination of different processes and analyses that receive as input a complete program and eventually produce a termination result. Our goal now is to obtain a *modular* analysis framework for COSTA which is able to produce termination proofs by analyzing programs one method at a time. I.e., in order to analyze a method m, we analyze the code of m only and (re-)use the analysis results previously produced for the methods invoked by m.

The communication mechanism used for this work is based on *assertions*, which store the analysis results for those methods which have already been analyzed. Assertions are stored by COSTA in a file per class basis and they keep information regarding the different analyses performed by COSTA: nullity, sign, size, heap properties, and termination.

Same as analysis results, assertions are of the form $m{:}Pre \mapsto Post$, where Pre is the *precondition* of the assertion and $Post$ is the *postcondition*. The precondition states for which call pattern the method has been analyzed. It includes information regarding all domains previously mentioned except size, which is context-insensitive. Pre_D (resp., $Post_D$) denotes the information of the precondition (resp., postcondition) related to analysis domain D. For example, $Pre_{nullity}$ corresponds to the information related to nullity in the precondition Pre. The postcondition of an assertion contains the analysis results for all domains produced after analyzing method m. Furthermore, the assertion also states whether COSTA has proved termination for that method.

In addition to assertions inferred by the analysis, COSTA has been extended to handle assertions written by the user, namely *assumed* assertions. These assertions are relevant for the cases in which analysis is not able to infer some information of interest that we know is correct. This can happen either because the analyzer is not precise enough or because the code of the method is not available to the analyzer, as happens with *native* methods, i.e., those implemented at low-level and for which no bytecode is available. The user can add assumed assertions with information for any domain. However, for the experiments described in Section 6 assumed assertions have been added manually for providing information about termination only, after checking that the library specification provided by Sun is consistent with the assertion. In assumed assertions where only termination information is available, abstract interpretation-based analyses take $\top$ as the postcondition for the corresponding methods.

4.1 Modular Bottom-up Analysis

The analysis of a Java program using the modular analysis framework consists in analyzing each of the methods in the program, and eventually determining if

the program will terminate or not for a given call pattern. Analyzing a method separately presents the difficulty that, from the analysis point of view, the code to be analyzed is *incomplete* in the sense that the code for methods invoked is not available. More precisely, during analysis of a method m there may be calls $m'{:}CP$ and the code for m' is not available. Following the terminology in [14], we refer to determining the value of AP to be used for $m'{:}CP \mapsto AP$ as the *answer patterns problem*.

Several analysis domains existing in COSTA are context-sensitive, and all of them, except heap properties analysis, are monovariant. For simplicity, the modular analysis framework we present is monovariant as well. That means that at most one assertion $m{:}Pre \mapsto Post$ is stored for each method m. If there is an analysis result for m', $m'{:}Pre \mapsto Post$, such that CP is applicable, that is, $CP \sqsubseteq Pre_D$ in the domain D of interest, then $Post_D$ can be used as answer pattern for the call to method m' in m.

For applying this schema, it is necessary that all methods invoked by m have been analyzed already when analyzing method m. Therefore, the analysis must perform a bottom-up traversal of the call graph of the program. In order to obtain analysis information for m' which is applicable during the analysis of m, it is necessary to use a call pattern for m' in its precondition such that it is equal or more general than the pattern actually inferred during the analysis of m. We refer to this as the *call patterns problem*.

Solving the *call* and *answer patterns problems*. A possibility for solving the *call patterns problem* would be to make the modular analysis framework polyvariant: store all possible call patterns to methods in the program and then analyze those methods for each call pattern. This approach has two main disadvantages: on one hand, it is rather complex and inefficient, because all call patterns are stored and every method must be analyzed for all call patterns that appear in the program. On the other hand, it requires performing a fixpoint computation through the methods in the program instead of a single traversal of the call graph, since different call patterns for a method may generate new call patterns for other methods.

Another alternative is a context-insensitive analysis. All methods are analyzed using $\top$ as call pattern for all domains. In this approach, all assertions are therefore applicable, although in a number of cases $\top$ is too general as call pattern for some domains, and the information obtained is too imprecise.

The approach finally used in this work tries to find a balance between both approaches. A monovariant modular analysis framework simplifies a great deal the behavior of the modular analysis, since a single traversal of the call graph is required. In contrast, it is context-sensitive: instead of $\top$, a default call pattern is used, and the result of the analysis is obtained based on this pattern. This framework uses different values as call patterns, depending on the particular analysis being performed. The default call pattern for nullity and sign is $\top$. For Heap properties analysis, in cyclicity it is the pattern that indicates that no argument of the method is cyclic. For variable sharing, it is the one that states

that no arguments share. The default call patterns used for analyzing methods are general enough to be applicable to most invocations used in the libraries and in user programs, solving the *call patterns problem*. However, there can be cases in which the call pattern of an invocation from other method is not included in the default pattern, i. e., $CP \not\sqsubseteq Pre_D$. If the code of the invoked method is available, COSTA will reanalyze it with respect to $CP \sqcup Pre_D$, even though it has been analyzed before for the default pattern. If the code is not available, $\top$ is used as answer pattern. A potential disadvantage of this approach is that all methods are analyzed with respect to a default call pattern, instead of the specific call pattern produced by the analysis. This means that the analyses in COSTA could produce more precise results when applied non modularly, even though they are monovariant, and it represents a possible loss of precision in the modular analysis framework. Nonetheless, in the experiments performed in Section 6 no method has been found for which it was not possible to prove termination using modular analysis, but it was proved in the non-modular model.

Cycles in the call graph. Analyzing just a method at a time and (re-)using analysis information while performing a bottom-up traversal of the call graph only works under the assumption that there are no cyclic dependencies among methods. In the case where there are strongly connected components (SCCs for short) consisting of more than one method, we can analyze all the methods in the corresponding SCC simultaneously. This presents no technical difficulties, since COSTA can analyze multiple methods at the same time. In some cases, we have found large cycles in the call graph that require analyzing many methods at the same time. In that case a different approach has been followed, as explained in Section 6. Therefore, in COSTA we perform a SCC study first to decide whether there are sets of methods which need to be handled as a unit.

Field-Sensitive Analysis. In some cases, termination of a method depends on the values of variables stored in the heap, i.e., fields. COSTA integrates a field-sensitive analysis [5] which, at least in principle, is a global analysis and requires that the source code of all the program be available to the analyzer. Nevertheless, in order to be able to use this analysis in the modular setting, a preliminary adaptation of that analysis has been performed. The field-sensitive analysis in COSTA is based on the analysis of program fragments named *scopes*, and modelling those fields whose behaviour can be reproducible using local variables. Fields must satisfy certain conditions in order to be handled as local variables. As a first step of the analysis, related scopes are analyzed in order to determine the fields that are consulted or modified in each scope. Given a method for which performing field-sensitive analysis is required in order to prove termination, an initial approximation to the set of methods that need to be analyzed together is provided by grouping those methods that use the same fields. We have precomputed these sets of methods by means of a non-modular analysis. Since the implementation of this preanalysis is preliminary and can be highly optimized, the corresponding time has not been included in the experiments in Section 6.

5 Application of Modular Analysis to phoneME libraries

We have extended the implementation of COSTA for the modular analysis framework. In order to test its applicability, we have analyzed the core libraries of the phoneME project, a well-known open-source implementation of Java Micro Edition (JavaME). We now discuss the main difficulties associated to the analysis of libraries:

- *Entry points.* Whereas a self contained program has a single entry method (`main(String[])`), a library has many entry points that must be taken into account during the analysis.
- *It is designed to be used in many applications.* Each entry point must be analyzed with respect to a call pattern that represents any *valid* call from any program that might use it. By valid we mean that the call satisfies the precondition of the corresponding method.
- *Large code base.* A system library, especially in the case of Java, usually is a large set of classes that implement most of the features in the source language, leaving only a few specific functionalities to the underlying virtual machine, mainly for efficiency reasons or because they require low-level processing.
- *With many interdependencies.* It is usual that library classes are extensively used from within library code. As a result of this, library code contains a great number of interdependencies among the classes in the library. Thus, non-modular analysis of a library method often results in analyzing a large portion of the library code.
- *Implemented with efficiency in mind.* Another important feature of library code is that it is designed to be as efficient as possible. This means that readability and structured control flow is often sacrified for relatively small efficiency gains. Section 6 shows some examples in phoneME libraries.
- *Classes can be extended and methods overridden.* Using a library in a user program usually not only involves object creation and method invocation, but also library classes can be extended and library methods overridden.
- *Use of native code.* Finally, it is usual that a library contains calls to native methods, implemented in C or inside the virtual machine, and not available to the analyzer.

5.1 Some Further Improvements to COSTA

While trying to apply COSTA to the phoneME libraries, we have identified some problems which we discuss below, together with the solutions we have implemented. As mentioned above, our approach requires analyzing methods in reverse topological order of the call graph. For this purpose, we extended COSTA in order to produce the call graph of the program after transforming the bytecode to a CFG. The call graph shows the complex structure of the classes in phoneME libraries. Furthermore, apparently, some cycles among methods existed in some of the call graphs, mainly caused by virtual invocations. However,

we observed that some potential cycles did not occur in practice. In these cases, either nullity and sign analyses remove some branches if they detect that are unreachable, or COSTA proves termination when solving the binary clauses system. A few cases include a large cycle that involves many methods. Those cycles are formed by small cycles focused in few methods (basically from `Object`, `String` and `StringBuffer` classes), and a large cycle caused by virtual invocations from those methods. In order to speed up analysis, methods in small cycles have been analyzed at the same time, as mentioned above, and large cycles have been analyzed considering the modular, method at a time bottom up approach.

In addition, COSTA has been extended for a more refined control of which pieces of code we want to include or exclude from analysis. Now there are several *visibility* levels: `method`, `class`, `package`, `application`, and `all`. When `all` is selected, all related code is loaded and included in the RBR. In the other extreme, when `method` is selected only the current method is included in the RBR and only the corresponding assertions are available for other methods.

5.2 An Example of Modular Analysis of phoneME libraries

As an example of the modular analysis framework presented in this paper, let us consider the method `Class.getResourceAsStream` in the phoneME libraries. It takes a string with the name of a resource in the application jar file and returns an object of type `InputStream` for reading from this resource, or `null` if no resource is found with that name in the jar file. Though COSTA analyzes bytecode, we show below the corresponding Java source for clarity of the presentation:

```
public java.io.InputStream getResourceAsStream(String name) {
    try {
        if (name.length() > 0 && name.charAt(0) == '/') {
           name = name.substring(1);
        } else {
           String clName = this.getName();
           int idx = clName.lastIndexOf('.');
           if (idx >= 0)
              name = clName.substring(0, idx+1).replace('.', '/') + name;
        }
        return new com.sun.cldc.io.ResourceInputStream(name);
    } catch (java.io.IOException x) { return null; }
}
```

In the source code of this method there are invocations to eleven methods of different classes (in addition to the eight methods explicitly invoked in the method code, the string concatenation operator in line 9 is translated to a creation of a fresh `StringBuffer` object and invocations to some of its methods.)

If the standard, non-modular approach of analysis is used, the analyzer would load the code of this method and all related methods invoked. In this case, there are 65 methods related to `getResourceAsStream`, from which 10 are native methods. In fact, using this approach COSTA is unable to prove termination. Using modular analysis, the call graph is traversed bottom-up, analyzing each

method related to getResourceAsStream one by one. For example, the analysis of the methods invoked by getResourceAsStream has obtained the following information related to the nullity domain[1]:

Method	call	result
StringBuffer.toString()	n/a	nonnull
StringBuffer.append(String)	$\top$	nonnull
StringBuffer.<init>()V	n/a	n/a
String.replace(char,char)	$(\top,\top)$	nonnull
com.sun.cldc.io.ResourceInputStream.<init>(String)	nonnull	n/a
String.substring(int)	$\top$	nonnull
String.length()	n/a	$\top$
String.substring(int,int)	$(\top,\top)$	nonnull
String.charAt(int)	$\top$	$\top$

In this table, the call pattern refers to nullity information regarding the values of arguments and the result is related to the method return value. Despite of the call patterns generated by the analysis of getResourceAsStream shown above, when the bottom-up modular analysis computation is performed, all methods are analyzed with respect to the default call pattern $\top$. The analysis of getResourceAsStream uses the results obtained for those methods to generate the nullity analysis results for getResourceAsStream. The same mechanism is used for other domains: sign, size and heap related properties.

Finally, two native methods are invoked from getResourceAsStream (lastIndexOf and getName) that require assumed assertions. In this case, $\top$ is assumed as the answer pattern for those invocations.

5.3 Contracts for Method Overriding

As mentioned above, one of the most important features of libraries in OO languages is that classes can be extended by users at any point in time, including the possibility of overriding methods. This poses significant problems to modular static analysis, since classes and methods which have already been analyzed may be extended and overridden, thus possibly rendering the previous analysis information incorrect. Let us illustrate this issue with an example:

```
class A {
   void m(){/* code for A.m() */};
   void caller_m(){this.m();};
};
class B extends A {
   void m(){/* code for B.m() */};
};

class Example {
   void method_main(A a){
      a.caller_m();
   };
};
```

Here, there are three different classes: A, B, and Example. But for now, let us concentrate on classes A and Example only. If A is analyzed, the result obtained for caller_m depends directly on the result obtained for A.m (for instance,

[1] These analysis results have been obtained ignoring possible exceptions thrown by the Java virtual machine (e.g., no method found, unable to create object, etc.) for clarity of the presentation.

`caller_m` could be guaranteed to terminate under the condition that `A.m` terminates). Then, the class `Example` is analyzed, using the analysis results obtained for `A`. Let us suppose that analysis concludes that `method_main` terminates.

Now, let us suppose that `B` is added to the program. As shown in the example, `B` extends `A` and overrides `m`. Imagine now that the analysis concludes that the new implementation of `m` is not guaranteed to terminate. The important point now is that the termination behavior of some of the methods we have already analyzed can be altered, and we have to make sure that analysis results can correctly handle this situation. In particular, `caller_m` is no longer guaranteed to terminate, and the same applies to `method_main`. Note, however, that class inheritance is correctly handled by the analyzer if all the code (in this case the code of `B`) is available from the beginning. This is done by considering, at the invocation program point, the information about both implementations of `m`. However, in general, the analyzer does not know, during the analysis of `A`, that the class will be extended by `B`. Such a situation is very common in the analysis of libraries, since they must be analyzed without knowing which user-defined classes will override their methods. In this example, corresponds to `A` and `Example` being library classes and `B` being defined by the user.

In order to avoid this kind of problems, the concept of *contract* can be used (in the sense of *subcontracting* of [20]). This means that the analysis result for a given method `m` is taken as the contract for m, i.e., information about how `m` and any redefinition of it is supposed to behave with respect to the analysis of interest. A *contract*, same as an assertion, has two parts: the calling *preconditions* which must hold in order the contract can be applicable; and the *postcondition*, the result of the analysis with respect to that preconditions. For example, a contract for `A.m()` may say that it terminates under the condition that the *this* object of type `A` is an acyclic data structure. In the example above, when `B` is added to the program, we have to analyze `B.m` taking as call pattern the precondition (Pre) in the contract for `A.m`. This guarantees that the result obtained for `B.m` will be valid in the same set of input states as the contract for `A.m`. Then, we need to compare the postconditions. If $m_B{:}Pre \mapsto Post^B$ and $m_A{:}Pre \mapsto Post^A$ are the assertions generated for `B.m` and `A.m`, respectively, and Pre is the default calling pattern for both implementations, there are two possible cases: (a) If $Post^B \sqsubseteq Post^A$ then `B.m` satisfies the contract for `A.m`; (b) otherwise, the contract cannot be applied, and `B.m` is considered incorrect. The user can manually inspect the code of `B.m` and if the analyzer loses precision, add an assumed assertion for `B.m`. Interfaces and abstract methods are similar to overriding methods of a superclass, with the difference that there is no code to analyze in order to generate the contract. In this case, assumed assertions written by the user can be used as contracts.

6 Experiments

After obtaining the call graph for the classes of phoneME's java.lang package, a bottom-up traversal of the call graphs has been performed. In a few particular

Class	Modular				Non Modular			Assumed		Related	
	$\#B_c$	#T	T_{cg}	$Time_a$	$\#B_c$	#T	$Time_a$	Nat	NNat	1st	All
Boolean	56	6	0.02	0.19	67	6	0.22	0	0	1	1
Byte	59	7	0.40	0.22	1545	7	21.10	0	0	4	22
Character	64	11	0.16	0.27	513	11	1.03	0	0	6	11
Class	110	4	1.17	1.10	4119	3	842.70	11	1	20	58
Double	107	17	3.66	1.12	107	13	0.36	2	0	8	57
Error	7	2	0.02	0.04	60	2	0.12	0	0	2	4
FDBigInt	1117	14	0.80	16.10	2513	12	158.39	0	2	23	47
Float	106	18	3.74	1.16	3105	15	5674.96	2	0	9	60
FloatingDecimal	3028	12	4.32	1201.10	3402	9	4983.88	0	8	49	64
Integer	469	21	1.35	18.76	4519	21	62.51	0	0	7	20
Long	268	11	0.64	10.99	2164	11	36.08	0	0	7	20
Math	207	16	0.14	0.67	212	16	0.69	6	0	3	3
NoClassDefFoundError	7	2	0.02	0.04	108	2	0.13	0	0	2	6
Object	737	3	0.21	46.21	891	3	129.31	5	0	7	28
OutOfMemoryError	7	2	0.02	0.03	170	2	0.18	0	0	2	8
Runtime	14	3	0.02	0.08	27	3	0.08	4	0	1	1
Short	59	7	0.39	0.24	1545	7	20.83	0	0	4	22
String	1784	39	5.88	21.11	8709	32	7217.43	6	3	34	120
StringBuffer	1509	37	6.74	11.01	14206	33	12103.35	0	0	37	86
System	45	7	0.38	0.31	2778	6	4864.33	5	0	11	62
Throwable	615	4	0.16	1.23	628	4	60.54	2	0	6	22
VirtualMachineError	7	2	0.02	0.04	108	2	0.14	0	0	2	6
Exception Classes (18)	136	38	0.61	0.74	3961	38	21.27	0	0	11	18
com/sun/* (7)	1584	26	5.55	22.36	11293	16	5161.29	0	0		
java/io/* (8)	106	11	1.47	0.65	2337	9	4983.35	0	0		
java/util/* (3)	265	13	0.88	3.33	2171	12	51.93	0	0		
Total	**12473**	**333**	**38.77**	**1359.10**	**71258**	**295**	**46396.17**	**43**	**14**	**256**	**746**

Table 1. Termination Analysis for java.lang package in COSTA (execution times are in seconds).

cases, it was required to enable other analyses included in COSTA (e.g., field sensitive analysis [5], as mentioned above) for proving termination, or disabling some features such as handling jvm exceptions.

Table 1 shows the results of termination analysis of java.lang package, plus some other packages used by java.lang. This table compares the analysis using the modular analysis described in this paper with the non-modular analysis previously performed by COSTA. The columns under **Modular** show the modular analysis results, while under the **Non Modular** heading non-modular results are shown. $\#\mathbf{B}_c$ shows the number of bytecode instructions analyzed for all methods in the corresponding class, **#T** shows the number of methods of each class for which COSTA has proved termination and $\mathbf{Time}_a$ shows the analysis time of all the methods in each class. In the modular case, the total analysis time is $\mathbf{Time}_a$ plus $\mathbf{T}_{cg}$, the time spent building the call graph of each class.

The two columns under **Assumed** show the number of methods for which assumed assertions were required: **Nat** is the number of native methods in each class, and **NNat** contains the number of non-native methods that could not be proved terminating. Finally, the last two columns under **Related** contain the number of methods from other classes that are invoked by the methods in the class, either directly, shown in **1st** or the total number of methods transitively invoked, shown in **All**. Some rows in the table contain results accumulated for

a number of classes (in parenthesis). The last three rows in the table contain accumulated information for methods directly or transitively invoked by the java.lang package which belong to phoneME packages other than java.lang.These rows do not include information about **Related** methods, since they are already taken into account in the corresponding columns for java.lang classes. The last row in the table, **Total**, shows the addition for all classes of all figures in each column. A number of interesting conclusions can be obtained from this table. Probably, the most relevant result is the large difference between the number of bytecode instructions which need to be analyzed in the modular and non-modular cases: 12,473 vs 71,258 instructions, i.e. nearly 7 times more code needs to be analyzed in the non-modular approach. The reason for this is that though in the modular approach methods are (at least in principle) analyzed just once, in the non-modular approach methods which are required for the analysis of different initial methods are analyzed multiple times. Obviously, this difference in code size to be analyzed has a great impact on the analysis times: the **Total** row shows that the modular analysis of all classes in java.lang is more than 30 times faster than the non-modular case.

Another crucial observation is that by using the modular approach we have been able to prove termination of 38 methods for which the non-modular approach is not able, either because the analysis runs out memory or because it fails to produce results within a reasonable time. Furthermore, the modular approach in this setting has turned out to be strictly more precise than the non-modular approach, since for all cases where the non-modular approach has proved termination, it has also been proved by the modular approach. This results in 333 methods for which termination has been proved in the modular approach, versus 295 in the non-modular approach. Altogether, in our experiments we have tried to prove termination of 389 methods. In the studied implementation of JavaME, 43 of those methods are native. Therefore, COSTA could not analyze them, and assumed assertions have been added for them. In addition, COSTA was not able to prove termination of 14 methods, neither in the modular nor non-modular approaches, as shown in the **NNat** column. For these methods, assumed assertions have also been added, and have not been taken into account in the other columns except in the last two ones. These two columns provide another view on the difference between using modular and non-modular analyses with respect to the number of transitively invoked methods (746) that required analysis, w.r.t. those directly invoked (256). In the modular case, only directly invoked methods need to be considered, and only for loading their assertions, whereas the non-modular approach requires loading (and analyzing) all related methods. We now describe in more detail the methods whose termination has not been proved by COSTA and the reasons for this:

- **Bitwise operations.** The size analysis currently available in COSTA is not capable of tracking numeric values after performing bitwise operations on them. Therefore, we cannot prove termination of some library methods which perform bitwise operations (in most cases, right or left shift operations) on variables which affect a loop termination condition.

– **Arrays issues.** During size analysis, arrays are abstracted to their size. Though this is sufficient for proving termination of many loops which traverse arrays, termination cannot be proved for loops whose termination depends on the value of specific elements in the array, since such values are lost by size abstraction.
– **Concurrency.** Though it is the subject of ongoing work, COSTA does not currently handle concurrent programs. Nonetheless, it can handle Java code in which `synchronized` constructs are used for preventing thread interferences and memory inconsistencies. In particular, few `java.lang` phoneME classes make real use of concurrency. For this reason, `Thread` class has not been included in the test, neither Table 1 does include information regarding `Class.initialize` nor `wait` methods defined in `Object`.
– **Unstructured control flow.** There are some library methods in which the control flow is unstructured, apparently for efficiency reasons. For example, `String.indexOf` uses a *continue* statement wrapping several nested loops, the outer most of them being an endless loop as in the following code (on the left):

```
indexOf(String str, int i){
 ...
 searchChar:
  while (true) {
   ...
   if (i > max) return -1;
   while (j < end) {
    if (v1[j++] != v2[k++]){
     i++; continue searchChar;}}
   return i - offset;} }
```

```
fixResourceName(String n){
 int stI = 0;
 int e = 0;

 while((e=n.indexOf('/',stI))!= -1){
  if (e == stI) {
   stI++;  continue;}
    .... } } }
```

– **Other Cases.** `ResourceInputStream.fixResourceName` involves a call to a native method in the loop condition (see code above on the right). A termination assertion is not enough to find a ranking function of the loop to prove termination.

7 Discussion

Modular analysis has received considerable attention in different programming paradigms, ranging from, e.g., logic programming [14, 11, 8] to object-oriented programming [22, 6, 19]. A general theoretical framework for modular abstract interpretation analysis was defined in [13], but most of the existing works regarding modular analysis have focused on specific analyses with particular properties and using more or less ad-hoc techniques. A previous work from some of the authors of this paper presents and empirically tests a modular analysis framework for logic programs [14, 11]. There are important differences with this paper: in addition to the programming paradigm, the framework of [14] is designed to handle one abstract domain, while the framework presented in this paper handles several domains at the same time, and the previous work is based on `CiaoPP`, a polyvariant context-sensitive analyzer in which an intermodular fixpoint algorithm was performed. In [22] a control-flow analysis-based technique is proposed

for call graph construction in the context of OO languages. Although there have been other works in this area, the novelty of this approach is that it is context-sensitive. Also, [6] shows a way to perform modular class analysis by translating the OO program into *open* DATALOG programs. In [19] an abstract interpretation based approach to the analysis of class-based, OO languages is presented. The analysis is split in two separate semantic functions, one for the analysis of an object and another one for the analysis of the context that uses that object. The interdependence between context and object is expressed by two mutually recursive equations. In addition, it is context-sensitive and polyvariant. As conclusion, in this work we have presented an approach which is, to the best of our knowledge, the first modular termination analysis for OO languages. Our approach is based on the use of assertions as communication mechanism between the analysis of different methods. The experimental results show that the approach increases the applicability of termination analysis. The flexibility of this approach allows a higher level of scalability and makes it applicable to component-based systems, since is not required that all code be available to the analyzer. Furthermore, the specification obtained for a component can be reused for any other component that uses it. It remains as future work to extend the approach to other intermediate cases between modular and global analysis, i.e., by allowing analysis of several methods as one unit, even if they are not in the same cycle. This can be done without technical difficulties and it should be empirically determined what granularity level results in more efficient analysis.

Acknowledgments

The authors would like to thank Damiano Zanardini for interesting discussions and for his help with the heap analysis in COSTA. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP'07*, LNCS, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO'07*, number 5382 in LNCS, pages 113–133. Springer, 2008.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource Usage Analysis and its Application to Resource Certification. In *FOSAD 2007/2008/2009 Tutorial Lectures*, LNCS 5705, pages 258–288. Springer, 2009.

5. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Diana Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *SAS 2010 Proceedings*, LNCS. Springer, 2010.
6. F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis, SAS 2003*, number 2694 in LNCS. Springer, 2003.
7. A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
8. M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. POPL'93*, 1993.
9. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.
10. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
11. J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *LOPSTR'05*, number 3901 in LNCS, pages 163–178. Springer-Verlag, April 2006.
12. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
13. P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Compiler Construction*, 2002.
14. G. Puebla et al. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, August 2004.
15. Samir Genaim and Damiano Zanardini. The acyclicity inference of COSTA. In *11th International Workshop on Termination*, July 2010.
16. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
17. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *POPL'01*, pages 81–92. ACM, 2001.
18. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
19. Francesco Logozzo. Separate Compositional Analysis of Class-based Object-oriented Languages. In *AMAST'2004*, volume 3116 of *LNCS*, pages 332–346. Springer-Verlag, July 2004.
20. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.
21. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Termination Analysis of Java Bytecode by Term Rewriting. In Johannes Waldmann, editor, *WST'09*, Leipzig, Germany, June 2009.
22. Christian W. Probst. Modular Control Flow Analysis for Libraries. In *Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, pages 165–179. Springer-Verlag, 2002.
23. F. Spoto, P.M. Hill, and E. Payet. Path-length analysis of object-oriented programs. In *EAAI'06*, ENTCS. Elsevier, 2006.
24. F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.
25. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM TOPLAS*, 32(3), 2010.