Termination and Cost Analysis with COSTA and its User Interfaces

E. Albert[1] P. Arenas[1] S. Genaim[1] M. Gómez-Zamalloa[1]
G. Puebla[2] D. Ramírez[2] G. Román[2] D. Zanardini[2]

[1] *DSIC, Complutense University of Madrid,*
{`elvira,puri,samir.genaim,mzamalloa`}`@fdi.ucm.es`

[2] *Technical University of Madrid,* {`german,diana,groman,damiano`}`@clip.dia.fi.upm.es`

Abstract

COSTA is a static analyzer for Java bytecode which is able to infer cost and termination information for large classes of programs. The analyzer takes as input a program and a resource of interest, in the form of a cost model, and aims at obtaining an upper bound on the execution cost with respect to the resource and at proving program termination. The COSTA system has reached a considerable degree of maturity in that (1) it includes state-of-the-art techniques for statically estimating the resource consumption and the termination behavior of programs, plus a number of specialized techniques which are required for achieving accurate results in the context of object-oriented programs, such as handling numeric fields in value analysis; (2) it provides several non-trivial notions of cost (resource consumption) including, in addition to the number of execution steps, the amount of memory allocated in the heap or the number of calls to some user-specified method; (3) it provides several user interfaces: a classical command line, a Web interface which allows experimenting remotely with the system without the need of installing it locally, and a recently developed Eclipse plugin which facilitates the usage of the analyzer, even during the development phase; (4) it can deal with both the Standard and Micro editions of Java. In the tool demonstration, we will show that COSTA is able to produce meaningful results for non-trivial programs, possibly using Java libraries. Such results can then be used in many applications, including program development, resource usage certification, program optimization, etc.

Keywords: Cost Analysis, Termination Analysis, Resource Usage.

⋆ This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS*

1 Introduction and System Description

We start by describing the architecture of COSTA, an abstract-interpretation-based static analyzer for studying the *cost* [4] and *termination* [1] behavior of Java bytecode [7] programs. *Cost analysis* deals with statically estimating the amount of *resources* which can be consumed at runtime (i.e., the cost), given the notion of a specific resource of interest, while the goal of *termination analysis* is to prove, when it is the case, that a program terminates for every input.

The input provided to the analyzer consists of a program and a description of the resource of interest, which we refer to as *cost model*. COSTA tries to infer an *upper bound* of the resource consumption, and *sound information* on the termination behavior (i.e., if the system infers that the program terminates then it should definitely terminate). The system comes equipped with several notions of cost, such as the *heap consumption*, the *number of bytecode instructions* executed, and the *number of calls* to a specific method.

COSTA is based on the classical approach to static cost analysis [14] which consists of two phases. First, given a program and a description of the resource, the analysis produces *cost relations*, which are sets of recursive equations. Second, closed-form solutions are found, if possible. For this, COSTA uses PUBS [2].

Having both cost and termination analysis in the same tool is interesting since such analyses share most of the computing machinery, and thus a large part of the analyzer is common to both. As an example, proving termination needs reasoning about the number of iterations of every loop in the program, which is also an essential piece of information for computing its cost.

In spite of being still a prototype, COSTA includes state-of-the-art techniques for cost and termination analysis, plus a number of specialized components and auxiliary static analyses which are required in order to achieve accurate results in the context of *object-oriented* programs, such as handling *numeric fields* in value analysis. As for the usability, the system provides several *user interfaces*: (i) a classical *command-line interface* (Section 2.1); (ii) a *Web interface* which allows using COSTA from a remote location, without the need of installing it locally (Section 2.2), and permits to upload user-defined examples as well as testing programs from a representative set; and (iii) a recently developed *plugin* for the widely used programming environment Eclipse [6], which allows easily using the analyzer while developing software (Section 2.3). COSTA can deal with full sequential Java, either in the *Standard Edition* [13] or the *Micro Edition* [8]. Needless to say, the analyzer works on Java bytecode programs, and does not require them to come from the compilation of Java source code: instead, bytecode may have been implemented by hand, or obtained by compiling languages different from Java.

The *tool demonstration* will show that COSTA is able to read `.class` files and produce meaningful and reasonably precise results for non-trivial programs, possibly using Java *libraries*. Possible uses of such cost and termination results include:

- helping the programmer in the *development* process, as obtained by using COSTA from the Eclipse plugin;
- the COSTA results can be used as guarantees that the program will not take too much time or resources in its execution nor fail to terminate; furthermore, this can potentially be combined with the *Proof-carrying code* [10] paradigm by adding certificates to programs which make checking resource usage more efficient.
- program *optimization*, COSTA can be used for guiding program optimization or choosing the most efficient implementation among several alternatives.

The preliminary experimental results performed to date are very promising and they suggest that resource usage and termination analysis can be applied to a realistic object-oriented, bytecode programming language.

2 User Interfaces of COSTA

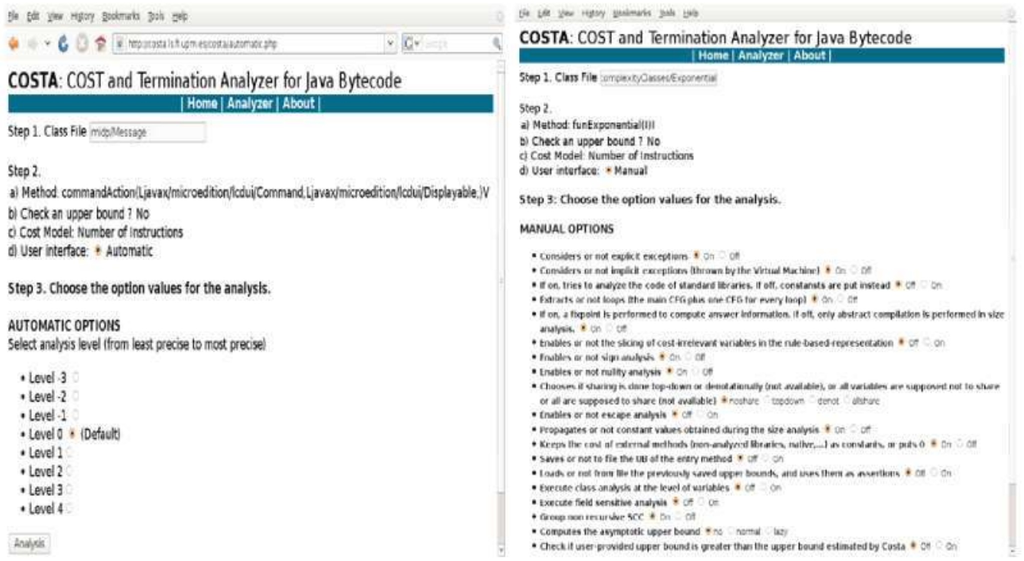
2.1 Command-Line Interface

COSTA has a command-line interface for executing COSTA as a standalone application. Different switches allow controlling the different options of the analyzer. It facilitates the implementation of other interfaces, as discussed below. They collect user information and interact with COSTA by making calls to its command-line interface.

2.2 Web Interface

The COSTA web interface allows users to try out the system on a set of representative examples, and also to upload their own programs, which can be in the form of either Java source, or as Java bytecode, in which case it can be given as a `.class` or a `.jar` file. As the behavior of COSTA can be customized using a relatively large set of options, the web interface allows two alternatives modes of use.

The first alternative, which we call *automatic* (see Figure 1, left) allows the user to choose from a range of possibilities which differ in the analysis


File Edit View History Bookmarks Tools Help
http://costa.ls.fi.upm.es/costa/automatic.php
| Home | Analyzer | About |
Analysis

COSTA: COST and Termination Analyzer for Java Bytecode

Step 1. Class File midp/Message

Step 2.
a) Method: commandAction(Ljavax/microedition/lcdui/Command,Ljavax/microedition/lcdui/Displayable;)V
b) Check an upper bound ? No
c) Cost Model: Number of Instructions
d) User interface: [✓ Automatic]

Step 3. Choose the option values for the analysis.

AUTOMATIC OPTIONS
Select analysis level (from least precise to most precise)

- Level -3
- Level -2
- Level -1
- Level 0 [✓] (Default)
- Level 1
- Level 2
- Level 3
- Level 4

COSTA: COST and Termination Analyzer for Java Bytecode

Step 1. Class File complexityClasses/Exponential

Step 2.
a) Method: funExponential(I)I
b) Check an upper bound ? No
c) Cost Model: Number of Instructions
d) User interface: [✓ Manual]

Step 3: Choose the option values for the analysis.

MANUAL OPTIONS

- Considers or not explicit exceptions [✓ On] Off
- Considers or not implicit exceptions (thrown by the Virtual Machine) [✓ On] Off
- If on, tries to analyze the code of standard libraries. If off, constansts are put instead [✓ Off] On
- Extracts or not loops (the main CFG plus one CFG for every loop) [✓ On] Off
- If on, a fixpoint is performed to compute answer information. If off, only abstract compilation is performed in size analysis. [✓ On] Off
- Enables or not the slicing of cost-irrelevant variables in the rule-based-representation [✓ Off] On
- Enables or not sign analysis [✓ On] Off
- Enables or not nullity analysis [✓ On] Off
- Chooses if sharing is done top-down or denotationally (not available), or all variables are supposed not to share or all are supposed to share (not available) [✓ noshare] topdown denot allshare
- Enables or not escape analysis [✓ Off] On
- Propagates or not constant values obtained during the size analysis [✓ On] Off
- Keeps the cost of external methods (non-analyzed libraries, native,...) as constants, or puts 0 [✓ On] Off
- Saves or not to file the UB of the entry method [✓ Off] On
- Loads or not from file the previously saved upper bounds, and uses them as assertions [✓ Off] On
- Execute class analysis at the level of variables [✓ Off] On
- Execute field sensitive analysis [✓ Off] On
- Group non recursive SCC [✓ On] Off
- Computes the asymptotic upper bound [✓ no] normal lazy
- Check if user-provided upper bound is greater than the upper bound estimated by Costa [✓ Off] On

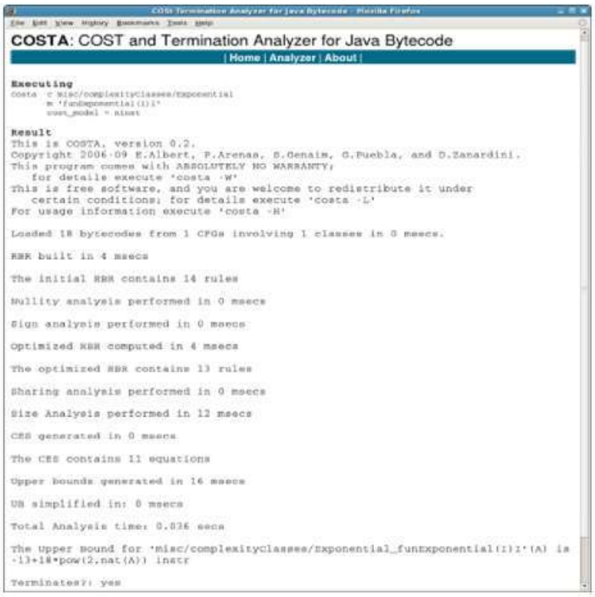
Fig. 1. Two ways of setting values for analysis options

accuracy and overhead. Starting from level 0, the default, we can increase the analysis accuracy (and overhead) by using levels 1 through 3. We can also reduce analysis overhead (and accuracy) by going down to levels -1 through -3. The main advantage of the automatic mode is that it does not require the user to understand the different options implemented in the system and their implications in analysis accuracy and overhead. The second alternative is called *manual* (see Figure 1, right) and it is meant for expert users. There, the user has access to all the analysis options, allowing a fine-grained control over the behavior of the analyzer. For instance, these options allow deciding whether to analyze the Java standard libraries or not, whether to take exceptions into account, to perform or not a number of pre-analyses, to write/read analysis results to file in order to reuse them in later analyses, etc.

Figure 2 shows the output of COSTA on an example program with exponential complexity. In addition to showing the result of termination analysis and an upper bound on the execution cost, COSTA (optionally) displays information about the time required by the intermediate steps performed by the analyzer in previous phases.

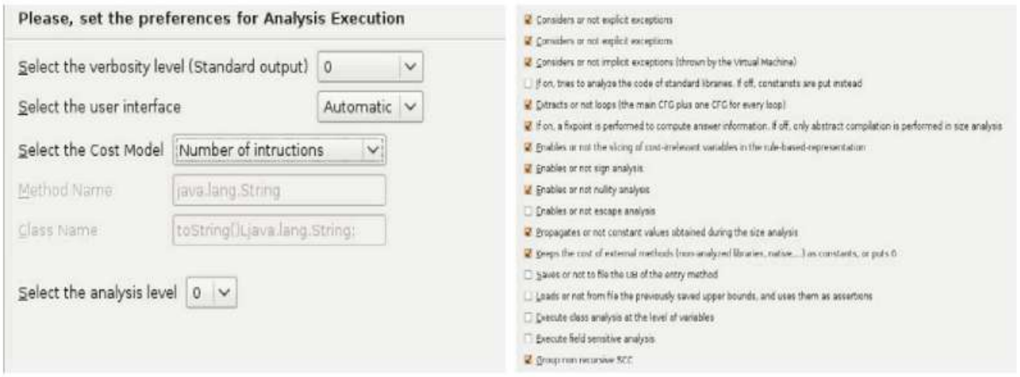
2.3 Eclipse Plugin

COSTA also has available an Eclipse plugin interface, which is fully integrated within the Eclipse development environment. This plugin allows programmers to analyze methods during the development process. It loads the `classpath` established for the project and uses for analysis the same classes and libraries


COST Termination Analyzer for Java Bytecode - Mozilla Firefox
File Edit View History Bookmarks Tools Help
COSTA: COST and Termination Analyzer for Java Bytecode
| Home | Analyzer | About |

```
Executing
costa  c misc/complexityClasses/Exponential
       m 'funExponential(I)I'
       cost_model = ninst

Result
This is COSTA, version 0.2.
Copyright 2006-09 E.Albert, P.Arenas, S.Genaim, G.Puebla, and D.Zanardini.
This program comes with ABSOLUTELY NO WARRANTY;
   for details execute 'costa -W'
This is free software, and you are welcome to redistribute it under
   certain conditions; for details execute 'costa -L'
For usage information execute 'costa -H'

Loaded 18 bytecodes from 1 CFGs involving 1 classes in 0 msecs.

RBR built in 4 msecs

The initial RBR contains 14 rules

Nullity analysis performed in 0 msecs

Sign analysis performed in 0 msecs

Optimized RBR computed in 4 msecs

The optimized RBR contains 13 rules

Sharing anal9sis performed in 0 msecs

Size Analysis performed in 12 msecs

CES generated in 0 msecs

The CES contains 11 equations

Upper bounds generated in 16 msecs

UB simplified in: 0 msecs

Total Analysis time: 0.036 secs

The Upper Bound for 'misc/complexityClasses/Exponential_funExponential(I)I'(A) is
-13+18*pow(2,nat(A)) instr

Terminates?: yes
```

Fig. 2. Results



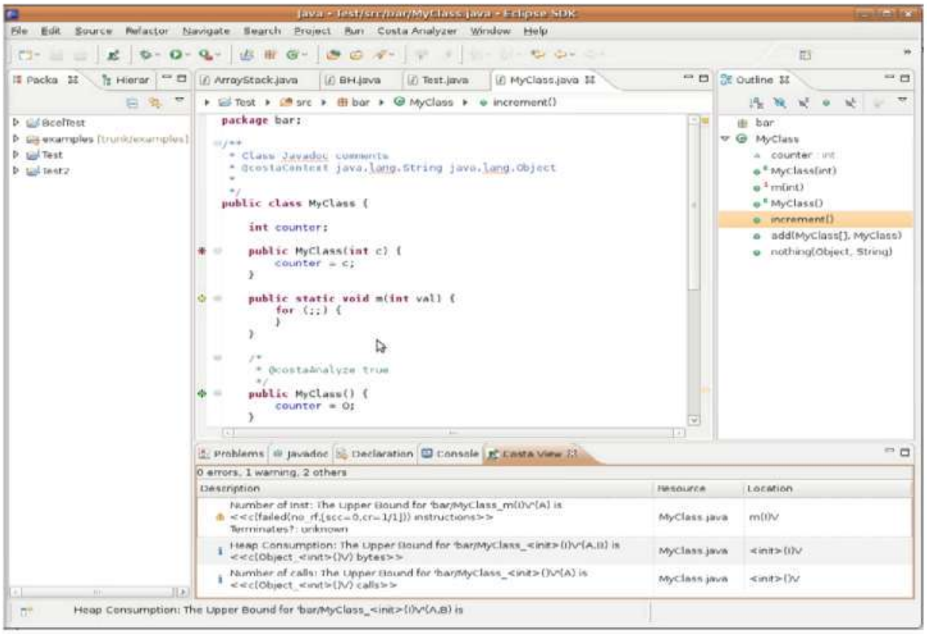
Please, set the preferences for Analysis Execution

Select the verbosity level (Standard output)	0
Select the user interface	Automatic
Select the Cost Model	Number of intructions
Method Name	java.lang.String
Class Name	toString()Ljava.lang.String;
Select the analysis level	0

- ✓ Considers or not explicit exceptions
- ✓ Considers or not explicit exceptions
- ✓ Considers or not implicit exceptions (thrown by the Virtual Machine)
- ☐ If on, tries to analyze the code of standard libraries. If off, constants are put instead
- ✓ Extracts or not loops (the main CFG plus one CFG for every loop)
- ✓ If on, a fixpoint is performed to compute answer information. If off, only abstract compilation is performed in size analysis
- ✓ Enables or not the slicing of cost-irrelevant variables in the rule-based-representation
- ✓ Enables or not sign analysis
- ✓ Enables or not nullity analysis
- ☐ Enables or not escape analysis
- ✓ Propagates or not constant values obtained during the size analysis
- ✓ Keeps the cost of external methods (non-analyzed libraries, native...) as constants, or puts 0
- ☐ Saves or not to file the UB of the entry method
- ☐ Loads or not from file the previously saved upper bounds, and uses them as assertions
- ☐ Execute class analysis at the level of variables
- ☐ Execute field sensitive analysis
- ✓ Group non recursive SCC

Fig. 3. COSTA Plugin Preferences

specified by the user to compile and execute the program. As in the web interface, users can configure a large set of options by using the Eclipse preferences configuration window, as shown in Fig. 3. These options are saved and loaded at every Eclipse execution. Also, the user can choose either the automatic analysis or the expert mode which allows a more fine-grained customization, like in the web interface. By using this plugin, one can analyze one or several methods from a class (see Fig. 5) or the whole class (by running the analysis on all its methods). The results of the analysis are shown using markers in the source code (see Fig. 4). Such markers are different depending on the cost model used for analysis. In addition, the plugin also shows all previous


Java - Test/src/bar/MyClass.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Costa Analyzer Window Help
Packa Hierar
ArrayStack.java BH.java Test.java MyClass.java
Outline
Test ▸ src ▸ bar ▸ MyClass ▸ increment()
BcelTest
examples [trunk/examples]
Test
Test2
bar
MyClass
counter : int
MyClass(int)
m(int)
MyClass()
increment()
add(MyClass[], MyClass)
nothing(Object, String)
Problems Javadoc Declaration Console Costa View

```java
package bar;

/**
 * Class Javadoc comments
 * @costaContext java.lang.String java.lang.Object
 *
 */
public class MyClass {

    int counter;

    public MyClass(int c) {
        counter = c;
    }

    public static void m(int val) {
        for (;;) {
        }
    }

    /*
     * @costaAnalyze true
     */
    public MyClass() {
        counter = 0;
    }
```

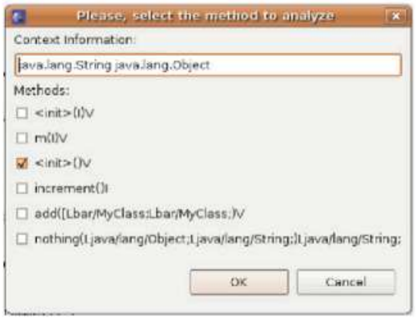
0 errors, 1 warning, 2 others

Description	Resource	Location
Number of inst: The Upper Bound for 'bar/MyClass_m(I)V'(A) is <<c(failed(no_rf,[scc=0,cr=1/1])) instructions>> Terminates?: unknown	MyClass.java	m(I)V
Heap Consumption: The Upper Bound for 'bar/MyClass_<init>(I)V'(A,B) is <<c(Object_<init>()V) bytes>>	MyClass.java	<init>(I)V
Number of calls: The Upper Bound for 'bar/MyClass_<init>()V'(A) is <<c(Object_<init>()V) calls>>	MyClass.java	<init>()V

Heap Consumption: The Upper Bound for 'bar/MyClass_<init>(I)V'(A,B) is

Fig. 4. COSTA Plugin Markers and View

analysis results in an additional view, which we call "the COSTA view". The COSTA view also includes a warning icon for methods whose termination is not proved, in order to alert the programmer about potential problems. It can also read comments in the source code, written in Javadoc style, in order to set up analysis information.



Context Information: java.lang.String java.lang.Object

Methods:

- [ ] <init>(I)V
- [ ] m(I)V
- [✓] <init>()V
- [ ] increment()I
- [ ] add([Lbar/MyClass;Lbar/MyClass;)V
- [ ] nothing(Ljava/lang/Object;Ljava/lang/String;)Ljava/lang/String;

Fig. 5. COSTA Plugin Methods Selection

3 Functionalities of COSTA

In this section, we explain the main functionalities of COSTA by means of several small examples. Some of these examples aim at illustrating the different cost

```
public static int funExp(int n) {
   if (n < 1) return 1;
   else return funExp(n - 1) + funExp(n - 2);
}
```

Fig. 6. Example for number of instructions

models available in the system. The last two examples are related to termination issues. In particular, we start in Sect. 3.1 by showing a program whose execution requires an exponential number of bytecode instructions. Then, in Sect. 3.2, we present the cost model that bounds the total heap consumption of executing a program and the recent extension to account for the effect of garbage collection. Sect. 3.3 performs resource analysis on a MIDlet using the cost model "number of calls" to a given method. Finally, in Sect. 3.4, we prove termination on an example whose resource consumption cannot be bound by COSTA and, also, show the latest progress to handle numeric fields(Sect. 3.5) in termination analysis.

3.1 Number of Instructions

The cost model which counts the number of instructions which are executed is probably the most widely used within cost analyzers, as it is a first step towards estimating the runtime required to run a program. Let us consider the Java method in Fig. 6. The execution of this method has an exponential complexity as each call spawns two recursive calls until the base case is found. COSTA yields the upper bound(slightly pretty printed) `-13 + 18*2`$^{\texttt{nat(n)}}$ using its automatic mode which indicates, as expected, that the number of instructions which are executed grows exponentially with the value of the input argument `n`. This shows that COSTA is not restricted to polynomial complexities, in contrast to many other approaches to cost analysis.

3.2 Memory Consumption

Let us consider the Java program depicted in Figure 7. It consists of a set of Java classes which define a linked-list data structure in an object-oriented style. The class `Cons` is used for data nodes (in this case integer numbers) and the class `Nil` plays the role of *null* to indicate the end of a list. Both `Cons` and `Nil` extend the abstract class `List`. Thus, a `List` object can be either a `Cons` or a `Nil` instance. Both subclasses implement a `copy` method which is used to clone the corresponding object. In the case of `Nil`, `copy` just returns a new instance of itself since it is the last element of the list. In the

```
abstract class List {
  abstract List copy();
}

class Nil extends List {
  List copy() {
    return new Nil();
  }
}

class Cons extends List {
  int elem;
  List next;
  List copy(){
    Cons aux = new Cons();
    aux.elem = m(this.elem);
    aux.next = this.next.copy();
    return aux;
  }

  static int m(int n) {
    Integer aux = new Integer(n);
    return aux.intValue();
  }
}  // class Cons
```

Fig. 7. Example for memory consumption

case of Cons, it returns a cloned instance where the data is cloned by calling the static method m, and the continuation is cloned by calling recursively the copy method on next.

The *heap* cost model of COSTA basically assigns, to each memory allocation instruction, the number of heap units it consumes. It can therefore be used to infer the total amount of memory allocated by the program. Running COSTA in automatic mode, level 0, yields the following upper bound for the copy method of class Cons:

$$\texttt{nat(this-1)*(12 + k}_1\texttt{ + k}_2\texttt{ + k}_3\texttt{) + 12 + 2*k}_1\texttt{ + k}_2\texttt{ + k}_3$$

It can be observed that the heap consumption is linear w.r.t. the input parameter **this**, which corresponds to the size of the *this* object of the method, i.e., the length of the list which is being cloned. This is because the abstraction being used by COSTA for object references is the *length of the longest reference chain*, which in this case corresponds to the length of the list. The expression also includes some constants. The symbolic constants k_1, k_2 and k_3 represent the memory consumption of the library methods which are transitively invoked. In particular, k_1 corresponds to the constructor of class Object and k_2 resp. k_3 to the constructor and intValue method of the class Integer. The numeric constant 12 is obtained by adding 8 and 4, being 8 the bytes taken by an instance of class Cons, and 4 the bytes taken by an Integer instance. Note that we are approximating the size of an object by the sum of the sizes of all of its fields. In particular, both an integer and a reference are assumed to consume 4 bytes.

Interestingly, we can activate the flag *go_into_java_api* and thus ask COSTA to analyze all library methods which are transitively invoked. In this case we obtain the upper bound 12*nat(this-1) + 12, for the same method. This is because the library methods used do not allocate new objects on the heap.

3.2.1 Peak Heap Consumption

In the case of languages with automatic memory management (*garbage collection*) such as Java Bytecode, measuring the total amount of memory allocated, as done above, is not very accurate, since the actual memory usage is often much lower. *Peak heap consumption analysis* aims at approximating the size of the *live* data on the heap during a program's execution, which provides a much tighter estimation. We have recently developed and integrated in COSTA a peak memory consumption analysis [5]. Among other things, this has required the integration of an *escape analysis* which approximates the objects which do not *escape*, i.e., which are not reachable after a method's execution. The upper bound `ub(A) = 8*nat(A-1) + 24` is now obtained for the same example.
An interesting observation is that the *Integer* object which is created inside the *m* method is not reachable from outside and thus can be garbage collected. The peak heap analyzer accounts for this and therefore deletes the size of the *Integer* object from the recursive equation, thus obtaining 8 instead of 12 multiplying $\mathtt{nat}(\mathtt{A}-\mathtt{1})$. By looking at the upper bound above, it can be observed that COSTA is not being fully precise, as the actual peak consumption of this method is $8 * \mathtt{nat}(\mathtt{A}-\mathtt{1}) + 8$ (i.e. the size of the cloned list). The reason for this is that the upper bound solver has to consider an additional case introduced by the peak heap analysis to ensure soundness, hence making the second constant increase to 24.

3.3 Number of Calls – Java Micro Edition

The Java Micro Edition (*Java ME*) [8] technology provides a limited environment to create Java applications which can be run on small devices with limited memory, display and power capacity. It is based on three elements: a *configuration* that provides the most basic set of libraries and virtual machine capabilities, a *profile* which is a set of APIs supported by mobile devices and an *optional package* (set of technology-specific APIs). MIDP (Mobile Information Device Profile) [12] is the profile that limits the set of APIs to only those functional areas considered as absolute requirements to achieve broad portability and successful deployments. A MIDlet is an application meeting the specifications for the Java ME technology, such as a game or a business application. Each MIDlet is an object of class MIDlet which follows a lifecycle [9], which is a state automaton managed by the Application Management System (*AMS*).

```
public void commandAction(Command c, Displayable s) {
 if (c == exitCommand) {
  destroyApp(false);
  notifyDestroyed();
 }
 if (c == sendMsgCommand) {
  try {
   TextMessage tmsg=(TextMessage)clientConn.newMessage(
                              MessageConnection.TEXT_MESSAGE);
   tmsg.setAddress("sms://+34697396559");
   tmsg.setPayloadText(msgToSend);
   clientConn.send(tmsg);
  }
  catch (Exception exc) {
   exc.printStackTrace();
  }
 }
}
```

Fig. 8. Example for number of calls

COSTA is able to perfom resource analysis on MIDlets by considering all classes used on each method called during the lifecycle of the MIDlet. Such methods are the constructor of the class, the `startApp()` and the `commandAction(Command c, Displayable d)` methods. In particular, the classes used during the analysis of the class constructor are added to the analysis of the `startApp()` method. After analyzing `startApp()` method, the current classes are used for analyzing the `commandAction(Command c, Displayable d)` method. As a result, the analyzer obtains a more precise cost and resource analysis for MIDP applications. Fig. 8 shows a simple but real example MIDlet that sends a text message: the text message is created (`newMessage` method), the recipient phone number set (`setAddress` method) and the text message is sent using the method `send(Message tmsg)` of the Wireless Messaging API.

We analyze this example using the cost model that counts the number of calls (*ncalls*) to a particular method. We apply it to obtain an upper bound on how many times the `send(Message tmsg)` method is called during the execution of `commandAction` method in a mobile device. COSTA outputs `1` as result, as it is to be expected.

```
static int factorial(int n) {
 int fact=1;
 for (int i=1; i<=n; i++) fact=fact*i;
 return fact;
};
```

```
static int doSum(List x) {
 if (x==null) return 0;
 else return factorial(x.elem)*doSum(x.next);
}
```

Fig. 9. Example for termination

3.4 Termination

Fig. 9 shows two methods which belong to the same class. The method `doSum` computes the sum of all factorial numbers contained in the elements of a linked list `x`, where `List` is defined as in Fig. 7. COSTA is able to ensure the termination of method `doSum` but no upper bound can be found by the system for the cost model *ninst*. The information that COSTA yields when computing an upper bound is:

```
The Upper Bound for 'doSum'(x) is nat(x)*(19+c(maximize_failed)*9)+4
Terminates?: yes
```

Intuitively, the cost of the calls to `factorial` cannot be bound because the value of `x.elem` is unknown at analysis time. However, we can still prove that the execution of the two methods always terminates by finding a so-called ranking function [11]. The technical details about how COSTA deals with termination can be found in [1].

3.5 Numeric Fields

Fig. 10 shows a Java program involving a numeric field in the condition of the loop of method `m`. This loop terminates in sequential execution because the field `size` is decreased at each iteration, at instruction `x.f.setSize(x.f.getSize() − 1)`, and, for any initial value of `size`, there are only a finite number of values which `size` can take before reaching zero. Unfortunately, applying standard value analyses on numeric fields can produce wrong results because numeric variables are stored in a shared mutable data structure, i.e., the heap. This implies that they can be modified using different references which are aliases and point to such memory location. Hence, further conditions are required to safely infer termination. COSTA incorporates a novel approach for approximating the value of heap allocated numeric variables [3] which greatly improves the precision over existing field-insensitive value analyses while introducing a reasonable overhead. For the example in Fig. 10, COSTA not only guarantees termination of method `m` but is also able to compute the (pretty printed) upper bound for `m(this,x,y,size) is 33+nat(size)*35` by using the cost model *ninst*.

```
class B {
 private int size;
 public int getSize(){return size;};
 public void setSize(int n){size=n;};
};
```

```
class A {
 private B f;

 int m(A x,B y) {
  int i=0;
  while (x.f.getSize()>0) {
   i=i+y.getSize();
   x.f.setSize(x.f.getSize()-1);
  }
  return i;
 }
};
```

Fig. 10. Example for termination in presence of numeric fields

4 Discussion and Future Work

costa is, to the best of our knowledge, the first tool for fully automatic cost analysis of object-oriented programs. Currently, the system can be tried online through the COSTA web site: `http://costa.ls.fi.upm.es`. We plan to distribute it soon under a GPL license. The fact that costa analyzes bytecode, i.e., compiled code, makes it more widely applicable, since it is customary in Java applications to distribute compiled programs, often bundled in jars, for which the Java source is not available.

As future work we plan to: (1) define new cost models to measure the consumption of new resources; (2) support other complexity schemes such as the inference of lower-bounds; (3) improve both the precision and performance of the underlying static analyses; and (4) handle the analysis of concurrent programs.

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, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*, LNCS 5079, 2008.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Dealing with numeric fields in termination analysis of java-like languages. In *FTfJP*, 2008.

[4] 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.

[5] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

[6] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* A-W, 1996.

[8] Java ME. `http://java.sun.com/javame/technology/index.jsp`.

[9] MIDP. `http://java.sun.com/javame/reference/apis/jsr118/javax/-microedition/midlet/package-summary.html`.

[10] G. Necula. Proof-Carrying Code. In *Proc. of ACM Symposium on Principles of programming languages (POPL)*, pages 106–119. ACM Press, 1997.

[11] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[12] Java Community Process MIDP Release. `http://jcp.org/aboutJava/communityprocess/final/jsr118-/index.html`.

[13] Java SE. `http://java.sun.com/javase/technologies/index.jsp`.

[14] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.