COSTABS: A Cost and Termination Analyzer for ABS *

Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa
Complutense University of Madrid, Spain
{elvira,puri,samir.genaim,mzamalloa}@fdi.ucm.es

Germán Puebla
Technical University of Madrid, Spain
german@fi.upm.es

Abstract

ABS is an *a*bstract *b*ehavioural *s*pecification language to model distributed concurrent systems. Characteristic features of ABS are that: (1) it allows abstracting from implementation details while remaining executable: a *functional sub-language* over abstract data types is used to specify internal, sequential computations; and (2) the *imperative sub-language* provides flexible concurrency and synchronization mechanisms by means of asynchronous method calls, release points in method definitions, and cooperative scheduling of method activations. This paper presents COSTABS, a COSt and Termination analyzer for ABS, which is able to prove termination and obtain *resource usage bounds* for both the imperative and functional fragments of programs. The resources that COSTABS can infer include termination, number of execution steps, memory consumption, number of asynchronous calls, among others. The analysis bounds provide formal guarantees that the execution of the program will never exceed the inferred amount of resources. The system can be downloaded as free software from its web site, where a repository of examples and a web interface are also provided. To the best of our knowledge, COSTABS is the first system able to perform resource analysis for a concurrent language.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D.1.3 [*Programming Techniques*]: [Concurrent Programming] *Distributed programming, Parallel programming*; D.3 [*Programming Languages*]: [Formal Definitions and Theory]

General Terms Languages, Theory, Verification, Reliability

Keywords Static Analysis, Resource Guarantees, Parallelism, Concurrent Objects

* 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 UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

1. Introduction

ABS [9] is an *a*bstract *b*ehavioral *s*pecification language for distributed object-oriented systems. Abstract behavioral specification languages can be situated between design-oriented and implementation-oriented specification languages. ABS addresses the specification of executable formal models for distributed object-oriented systems: it allows a high-level specification of a system, including its concurrency and synchronization mechanisms as well as local state updates. Thus, ABS models capture the concurrent control flow of object-oriented (OO) systems, yet abstract away from many implementation details which may be undesirable at the modeling level, such as the concrete representation of internal data structures, the scheduling of method activations, and the properties of the communication environment. Scheduling in ABS is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified.

Cost analysis [12] (a.k.a. resource usage analysis) aims at automatically inferring bounds on the resource consumption of programs statically, i.e., without having to execute the program. The inferred bounds are symbolic expressions given as functions of its *input data sizes*. For instance, given a method `void traverse(List l)`, an upper bound (e.g., for number of execution steps) can be an expression of the form `size(l)*200+10`, which guarantees that the number of steps will never exceed such amount. In recent work [1], we have presented a framework for performing cost analysis of ABS programs. The main novelties of [1] are related to the concurrency and distribution aspects of the language. (1) Concurrency poses new challenges to the process of obtaining sound and precise size relations. This is mainly because the interleaving behavior inherent to concurrent computations can influence how the sizes of data are modified. For instance, a class field which acts as loop counter can be decreased, by some interleaved concurrent task, while the task executing such loop is suspended and waiting for some condition to hold. (2) Distribution does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. [1] proposes the use of *cost centers* to keep the resource consumption of the different distributed components separate.

This paper presents COSTABS, an implementation of the cost analysis framework for concurrent objects introduced in [1]. The system incorporates a sound size analysis, whose precision can be increased by means of invariants on the class fields. For simple examples, the invariants can be automatically generated. In more complex cases, the user can provide the invariants by means of annotations in the source code. The system also allows computing the cost bound as a monolithic expression or, more interestingly, in separated cost centers. Performing cost analysis at the level of a specification language has as an additional advantage that performance errors can be detected early in the software devel-

opment process, e.g., we can observe bottlenecks in a distributed system if one component has a large resource consumption while siblings are idle most of the time. COSTABS can be used through three different interfaces: a command-line interface, a web interface and an Eclipse plugin. The system is open-source and can be downloaded (together with examples, documentation, etc.) from: `http://costa.ls.fi.upm.es/costabs`.

2. The ABS Language

The language ABS [9] is a successor of Creol [6, 10]. It is an OO language for distributed concurrent systems whose concurrency model is based on *concurrent objects*. An ABS *program* defines interfaces, classes, datatypes, and functions, and has a `main` block to configure the initial state. It has is a standard strict *functional sub-language* which is used to abstract from implementation details: abstract data types are used to specify internal, sequential computations. The *concurrent imperative sub-language* has Java-like syntax but with some important differences w.r.t. Java. One is that statements can only access fields of the *current class*. This means that a concurrent object can be run in a dedicated processor, as it encapsulates a local heap which is not accessible from outside the object. Other differences are that classes may contain parameters which are treated as standard class fields, and all methods return a value (Unit plays the role of void in Java).

The main difference with Java is on the concurrency model which is not thread-based but rather based on the notion of *concurrent objects*. Concurrent objects were introduced to provide programmers with simple language extensions which allow programming concurrent applications with relatively little effort and in a less error-prone way. In this model, communication is based on asynchronous method calls that can be synchronized by means of *future* variables as follows:

- Asynchronous method calls, denoted $x{=}o!m(\overline{e})$, allow the caller to proceed with its execution without blocking on the call. Here x is a future variable, o is an object (typed by an interface), and $\overline{e}$ are expressions which many involve functional data. A future variable x refers to a return value which has yet to be computed.
- External synchronization in ABS is controlled by means of two operations, `await` and `get`. In `await` g, the guard g is a Boolean condition. It is often used to test if a future variable has been bound to a value already, expressed by means of the test $x?$. If g evaluates to false, the processor is released, the current process is *suspended* and the processor becomes idle. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled.
- The other synchronization operation is `get`. It allows retrieving the return value by blocking all execution in the object until the return value is available.
- A *synchronous call*, abbreviated as $v{=}o.m(\overline{e})$, is internally transformed into the statement sequence $x{=}o!m(\overline{e})$; $v{=}x$.`get`.

EXAMPLE 2.1. *Our running example is the mail server application in Fig. 1. At the top, we see a fragment of the functional subprogram which includes type definitions (String is predefined) and function lookup. The imperative concurrent part contains the interfaces and the implementation of all classes. Calls to functions and functional data structures appear in italics. A mail server is composed of an address book (the class parameter* ab*) and a list of user names (the field* listUsers*). User names can be added to the server by invoking method* addUser*. Method* notify *sends a message (*m*) to all users in the list* listUsers*. To this end, it first asynchronously invokes* getUserAddress *in order to retrieve the next user (variable* u*) in the list. The* await *instruction allows releasing the processor if the information is not ready. The next instruction* get *blocks the execution of the current task until the requested information has arrived. When it arrives, the asynchronous call to* receive *is encharged of sending the message to the corresponding user without any kind of synchronization.*

```
data List⟨A⟩=Nil | Cons(A,List⟨A⟩);
data Pairs⟨A,B⟩=Pair(A,B);
data Map⟨A,B⟩=EmptyM |
       Assoc(Pairs⟨A,B⟩,Map⟨A,B⟩);
type UserName=String;
type Message=String;
def B lookup⟨A,B⟩(Map⟨A,B⟩ ms, A k) =
  case ms {Assoc(Pair(k,y),_) ⇒ y;
           Assoc(_,tm) ⇒ lookup(tm,k);}
interface AddressBook {
  User getUserAddress(UserName u);
}
interface User {
  Unit receive(Message msg);
}
interface MailServer {
  Unit addUser(UserName u);
  Unit notify(Message m);
}
class AddressBookImp implements AddressBook {
  Map⟨UserName,User⟩ users = EmptyM;
  User getUserAddress(UserName u){
    return lookup(users,u);
  }
}
class UserImp implements User {
  List⟨Message⟩ msgs = Nil;
  Unit receive(Message msg) {
    msgs = Cons(msg,msgs);
  }
}
class MailServerImp(AddressBook ab)
        implements MailServer {
  List⟨UserName⟩ listUsers = Nil;
  Unit addUser(UserName u) {
    listUsers = Cons(u, listUsers);
  }
  Unit notify(Message m) {
    while (listUsers != Nil) {
      Fut⟨User⟩ u;
      u = ab!getUserAddress(head(listUsers));
      await u ? ;
      User us = u.get;
      us!receive(m);
      listUsers = tail(listUsers);
    }
  }
}
```

Figure 1. ABS Implementation of a Mail Server

Due to the fact that objects encapsulate their own heaps, the language allows distribution and parallelism. In particular, each object (or group of objects [11]) can be potentially run in a separate processor in parallel with the others.

EXAMPLE 2.2. *Fig. 2 shows a graphical representation of the concurrent activity when executing the program with two users* us_1 *and* us_2*. It can be observed that the four objects (*us_1*,* us_2*,* ms *and* ab*) become distinct concurrent entities which communicate with each other by means of asynchronous calls (shown as labeled arrows)*

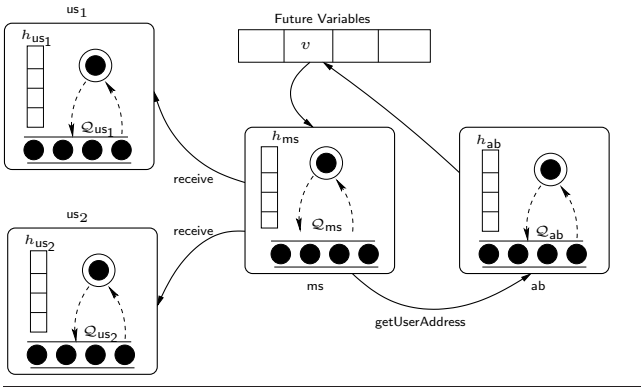

Figure 2. Overview of Concurrent Activity

and use future *variables (shown at the top) to eventually return/retrieve the results. Inside the objects, we depict the fact that concurrent objects have their own heaps h_o (not accessible from outside), their queue of pending tasks $\mathcal{Q}_o$ and an active task (if any).*

3. The Cost and Termination Analyzer

The process of inferring the resource consumption (or cost) of a given program consists of the following steps: (1) Selecting a cost model which determines the type of resource whose consumption we are interested in approximating; (2) Applying size analysis to infer information on how the sizes of the different data structures change during the execution; (3) Generating cost equations that describe the cost of the program in terms of its input data sizes and solving them into closed-form lower/upper bounds. These steps, in principle, are required for cost analysis of both sequential and concurrent programs, however, in the concurrent setting the technical details of each step are more complicated (see [1]). In what follows, we explain these steps on the running example, by laying stress on the differences w.r.t. cost analysis in a sequential setting.

3.1 Cost Models

Briefly, a cost model is a function that maps each instruction to the amount of resources consumed when executing it. COSTABS provides the following cost models, the first three ones are inherited from the sequential setting, while the last two ones are specific for concurrent programs.

- *Termination*: this model does not require assigning cost to instructions (or, what it is the same, it assigns them cost zero). Hence, as the accumulated cost is zero, in order to infer an upper bound, the analyzer just needs to prove that all loops in the program terminate.
- *Steps*: it tries to approximate the number of executed instructions, including both instructions of the imperative and the functional parts of the program.
- *Memory*: the memory consumption estimates the size of the terms constructed in the functional part of the language. This is because objects are meant to be the concurrency units, while the data structures are constructed using terms.
- *Objects*: it counts the total number of objects created along the execution. This provides an indication of the amount of parallelism that might be achieved, since each object could be running in a different a processor.
- *Task-level*: it estimates the number of tasks that are spawned along an execution. This can be counted by tracing how many asynchronous calls are performed. The task-level is useful for finding optimal deployment configurations, and detect situations like when one component is receiving too many requests while its siblings are idle.

In addition to the above models, it is easy to add new cost models to the system by just mapping each instruction to a corresponding cost according to the model.

3.2 Size Analysis

The objective of size analysis is to infer *size relations* which allow reasoning on how the sizes of data change along the program's execution. This information is essential, among other things, for bounding the number of iterations that loops perform. For example, for the while loop in method notify, size analysis infers that the size of the list listUsers decreases at each iteration, and thus the number of iterations is bounded by its initial size.

The first step in size analysis is to define the meaning of *size* of a term (i.e., the size of a data structure). For this, COSTABS relies on the notion of *norms* [3] which are functions that map terms to their sizes. By default, COSTABS uses the *term-size* norm, which counts the number of type constructors in a given term. Any norm can be used in the analysis, depending on the nature of the data structures used in the program. For instance, the *list-length* norm that counts the number of elements in lists, or the *term-depth* norm that calculates the depth of the corresponding data structures.

Once the norms are chosen, COSTABS applies a global analysis that infers relations between the sizes of the different variables at different program points, w.r.t. the chosen norm. This analysis, in principle, is done by means of a fixpoint computation over some numerical abstract domain. For the functional part, since it is sequential, existing size analysis for sequential settings can be applied [5]. For example, such analysis is able to infer that the size of the first argument of function *lookup* is decreasing when calling it recursively. For the imperative part, the analysis must be modified to handle the concurrency primitives, otherwise, soundness is not guaranteed mainly because such analysis does not account for modifications of the global state by other tasks.

COSTABS modifies a classical sequential size analysis in order to handle the concurrency primitives as follows: (a) when executing an instruction which does not cause the suspension of the current task, then fields (i.e., the global state) are tracked as if they were local variables, since in the concurrent objects setting it is guaranteed that in such circumstances no other tasks can modify those fields simultaneously; and (b) when executing an instruction that might cause suspension (e.g., `await`) of the current task, then the analysis loses all information about the corresponding fields, this is because they might be modified by other tasks in the meantime.

This simple modification guarantees soundness of size analysis for a concurrent setting. However, it often loses precision. For example, in the while loop of method notify, losing the information on the field listUsers when executing `await` prevents us from proving that its size decreases in each iteration. Thus, COSTABS fails to bound the number of iterations of that loop. To overcome this problem, COSTABS provides a way to incorporate class invariants that provide guarantees on the global states when the process is resumed. For example, if we add the following invariant (using JML syntax)

```
//@invariant \old(listUsers) == listUsers
```

before the `await` instruction in the while loop of method notify, then we state that it is guaranteed that when the process resumes, the value of listUsers will be the same as when the process has been suspended. Under these conditions, and taking into account the effect of the last instruction of the loop, it is possible to prove

that the size of listUsers decreases at each iteration, and thus bound the number of iterations.

3.3 Cost Centers

The last step in cost analysis uses the inferred size relations and the selected cost model in order to generate cost equations that capture the cost of the program in terms of its input, and solves them into closed-form bounds. These cost equations are similar to classical recurrence equations, but with multiple arguments and a high degree of non-determinism. (We skip the technical details, the interested reader can refer to [1].) Let us directly explain the upper bounds that we obtain for the running example.

By applying the analysis starting from method notify and using the *Steps* cost model, we obtain the following upper bound (after simplifying the constants for the sake of readability):

$$5 + (22 + 4 * users^{+}) * listUsers^{+}$$

Variables $listUsers^{+}$ and $users^{+}$ refer to the maximum sizes of the fields listUsers and users respectively. The subexpression $(22 + 4 * users^{+})$ refers to the cost of each iteration of the while loop. This includes the cost of the called methods and functions, namely getUserAddress, receive, and *lookup*, as well as the cost of the local instructions. Note that the subexpression $4 * users^{+}$ refers to the cost consumed by function *lookup*. The constant 4 is for executing the code of *lookup* once, and $users^{+}$ is the number of recursive calls. The cost of each iteration is then multiplied by $listUsers^{+}$, which is a bound on the number of iterations of the while loop. Finally, we add 5 to account for the cost of the instructions outside the loop (in this case it refers to the last comparison of the while loop's guard).

COSTABS includes also an option to split the cost into *Cost Centers* that represent the different distributed components of the system. This allows us to obtain the cost per component rather than a single cost expression as we have seen above. The current implementation of COSTABS assumes that objects of the same type belong to the same cost center, i.e., they share the processor. By applying the analysis with the same settings as above, but with the cost center option enabled, we obtain the upper bounds (after simplification of constants for the sake of readability):

Cost Center	Upper Bound
MailServerImp	$5 + 16 * listUsers^{+}$
UserImp	$3 * listUsers^{+}$
AddressBookImp	$(3 + 4 * users^{+}) * listUsers^{+}$

Observe that the sum of these bounds is identical to the single bound we have obtained before. The difference is that the new expressions provide additional information that indicates the cost per cost center as follows: (1) Cost center MailServerImp accounts for the cost of executing the local instructions of the while loop, which are $16 * listUsers^{+}$ steps, plus that of executing the code outside the loop, which is 5 steps; (2) Cost center UserImp consumes the cost of executing receive, which is 3 steps, $listUsers^{+}$ times; and (3) Cost center AddressBookImp includes the cost of executing getUserAddress and *lookup*, which is $3 + 4 * users^{+}$ steps, $listUsers^{+}$ times. By using cost centers, it is possible to observe that most of the work is done on cost center AddressBookImp, which has a quadratic complexity while the others are linear.

Similarly, by applying cost analysis for the *Task-Level* cost model (which counts the number of calls to methods) and, without separating in cost centers, we obtain the following upper bounds:

Method/Function	Upper Bound
notify	1
receive	$listUsers^{+}$
getUserAddress	$listUsers^{+}$

As regards the *Memory* cost model, COSTABS just infers that addUser has a constant consumption. The other parts of the code do not construct any term. Also, for the *Objects* cost model, we get cost zero, as no objects are created in the analyzed methods.

4. Conclusions and Future Work

We have presented COSTABS, a cost analyzer of ABS concurrent programs. All existing resource usage analysis tools to-date handle sequential code, namely SPEED [7] analyzes sequential C programs, RAML [8] functional programs and COSTA [2] sequential Java bytecode programs. The only common part between COSTA and COSTABS is that both systems use the same solver to generate upper bounds from the cost relations. However, the main part of the analysis which builds from the original program a system of cost relations is totally new and independent. Therefore, in spite of the name, COSTABS cannot be considered an extension of COSTA to handle concurrency, but rather a new system. In the context of termination analysis, Terminator [4] is the only tool to handle concurrent Java programs. COSTABS not only proves termination of programs, but it also solves the more difficult problem of generating upper bounds for their resource consumption.

We plan to improve COSTABS in two main directions. We want to infer the shape of the deployment configurations to determine the different cost centers that a particular system has, rather than assigning them to classes, as the system currently does. Also, we want to be able to infer the invariants on the class fields which are necessary to obtain the upper bounds in a fully automatic way. Currently, only very simple cases can be inferred automatically.

References

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, LNCS. Springer, 2011. To appear.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 2011. To appear.

[3] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In *Proc. of TAPSOFT'91*, LNCS 494, pages 153–180. Springer, 1991.

[4] B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *Proc. of PLDI'07*, pages 320–330. ACM, 2007.

[5] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proc. of POPL'78*, pages 84–97. ACM Press, 1978.

[6] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.

[7] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.

[8] M. Hofmann J. Hoffmann, K. Aehlig. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.

[9] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10*, LNCS. Springer, 2011. To appear.

[10] E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

[11] J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.

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