# Quantified Abstractions of Distributed Systems

Elvira Albert[1], Jesús Correas[1], Germán Puebla[2] and Guillermo Román-Díez[2]

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

**Abstract.** When reasoning about distributed systems, it is essential to have information about the different kinds of nodes which compose the system, how many instances of each kind exist, and how nodes communicate with other nodes. In this paper we present a static-analysis-based approach which is able to provide information about the questions above. In order to cope with an unbounded number of nodes and an unbounded number of calls among them, the analysis performs an *abstraction* of the system producing a graph whose nodes may represent (infinitely) many concrete nodes and arcs represent any number of (infinitely) many calls among nodes. The crux of our approach is that the abstraction is enriched with upper bounds inferred by a *resource analysis* which limit the number of concrete instances which the nodes and arcs represent. The combined information provided by our approach has interesting applications such as debugging, optimizing and dimensioning distributed systems.

## 1 Introduction

When reasoning about distributed systems, it is essential to have information about their *configuration*, i.e., the sorts and quantities of nodes which compose the system, and their *communication*, i.e., with whom and how often the different nodes interact. Whereas configurations may be straightforward in simple applications, the tendency is to have rather complex and dynamically changing configurations. Cloud computing [5] is an example of this. In this paper, we introduce the notion of *Quantified Abstraction* (QA for short) of a distributed system which abstracts both its configuration and communication by means of static analysis. QAs are *abstract* in the sense that a single abstract node may represent (infinitely) many nodes and a single abstract interaction may represent (infinitely) many interactions. QAs are *quantified* in that we provide an upper bound on the (possibly infinite) number of actual nodes which each abstract node represents, and an upper bound on the (possibly infinite) number of actual interactions which each abstract interaction represents. Note that abstraction allows dealing with an unbounded number of elements in the system, whereas the upper bounds allow regaining accuracy by bounding the number of elements which each abstraction represents.

Actors form a well established model for distributed systems [14,4,6,12]. We apply our analysis to an Actor-like language [10] for distributed concurrent systems based on asynchronous communication. The distribution model is based on (possibly interacting) objects which are grouped into distributed *nodes*, called *coboxes*. Objects belong to their corresponding cobox for their entire lifetime. To realize concurrency, each cobox supports multiple, possibly interleaved, processes which we refer to as *tasks*. Tasks are created when methods are asynchronously called on objects, e.g., $o!m()$ starts a new task. The callee object $o$ is responsible for executing the method call. The communication can be observed by tracking the calls between each pair of objects (e.g., we have a communication between the *this* object and $o$ due to the invocation of $m$). Informally, given an execution, its *configuration* consists of the set of coboxes which have been created along such execution and which are the nodes of the distributed system, together with the set of objects created within each cobox. Similarly, the *communication* of an execution is defined as the set of calls between each pair of objects in the system; from which we can later obtain the communication for pairs of coboxes.

Statically inferring QAs is a challenging problem, since it requires (1) keeping track of the relations between the coboxes and the objects, (2) bounding the number of elements which are created, (3) bounding the number of interactions between objects, and (4) doing so in the context of distributed concurrent programming. The main contributions of this paper are:

1. *Abstract configurations*. The abstraction of objects and coboxes we rely on is based on *allocation sequences* [11] (i.e., the sequence of allocation sites where the objects that led to the creation of the current one were created). We use a points-to analysis to infer the allocation sequences which allow us to infer the ownership relations between the coboxes and the objects created.
2. *Quantified nodes*. We define a new cost model which can be plugged in the generic resource analyzer COSTABS [2] (without requiring any change to the analysis engine) in order to infer upper bounds on the number of coboxes and of objects that each element of an abstract configuration represents.
3. *Quantified edges*. We propose a cost model which can be also plugged in COSTABS to infer upper bounds on the number of calls among nodes.
4. *Implementation*. We have implemented our analysis in COSTABS and applied it on a case study developed by Fredhopper®. A notable result of our experiments is that COSTABS was able to spot an excessive number of connections between two distributed nodes that should be better allocated together.

QAs have many applications for optimizing, debugging and dimensioning distributed applications which include among others: (1) QAs provide a global view of the distributed application, which may help to detect errors related to the creation of the topology or task distribution. (2) They allow us to identify nodes that execute a too large number of processes while other siblings execute only a few of them. (3) They are required to perform meaningful resource analysis of distributed systems, since they allow determining to which node the computation of the different processes should be associated. (4) They allow us to detect components that have many interactions and that would benefit from being deployed in the same machine or at least have a very fast communication channel. (5) They provide a further step towards static bandwidth analysis.

## 2 The Language

We apply our analysis to the language ABS [10,12]. ABS extends the basic concurrent objects model [14,4,6,12] with the abstraction of object groups, named *coboxes*. Each cobox conceptually has a dedicated processor and a number of objects can live inside the cobox and share its processor. Communication is based on asynchronous method calls with standard objects as targets. Consider an asynchronous method call $m$ on object $o$, written as $f = o!m()$. The objects *this* and $o$ communicate by means of the invocation $m$. Here, $f$ is a future variable which allows synchronizing with the completion of task $m$ by means of the await $f?$ instruction which behaves as follows. If $m$ has finished, execution of the current task proceeds. Otherwise, the current task releases the processor to allow other available tasks (possibly a task of another object in the cobox) to take it. The language syntax is as follows. A *program* consists of a set of classes **class** $C_1$ $(t_1\ fn_1,\ldots,t_n\ fn_n)$ $\{M_1\ \ldots\ M_k\}$ where each $t_i\,fn_i$ declares a field $fn_i$ of type $t_i$, and each $M_i$ is a *method definition* $t\ m(t_1\ w_1,\ldots,t_n\ w_n)\ \{t_{n+1}\ w_{n+1};\ldots;t_{n+p}\ w_{n+p};\ s\}$ where $t$ is the type of the return value; $w_1,\ldots,w_n$ are the formal parameters with types $t_1,\ldots,t_n$; $w_{n+1},\ldots,w_{n+p}$ are local variables with types $t_{n+1},\ldots,t_{n+p}$; $s$ is a sequence of instructions which adhere to the following grammar, where $x$ and $z$ denote standard variables, and $y$ a future variable whose declaration includes the type of the returned value:

$s ::= in \mid in; s \quad b ::= e{>}e \mid e{==}e \mid b{\wedge}b \mid b{\vee}b \mid !b \quad e ::= \text{null} \mid \text{this}.f \mid x \mid e{+}e \mid e{*}e \mid e{-}e$
$in ::= x{=}\text{new}\ C(\bar{x}) \mid x{=}\text{newcog}\ C(\bar{x}) \mid x{=}e \mid \text{this}.f{=}e \mid y = x!m(\bar{z}) \mid \text{if}\ b\ \text{then}\ s_1\ \text{else}\ s_2 \mid$
$\quad \text{return}\ x \mid \text{while}\ b\ \text{do}\ s \mid \text{await}\ y?$

There is an implicit local variable called this that refers to the current object. Observe that the only fields which can be accessed are those of the current object, i.e., this. Thus, the language is data-race free [10], since no two tasks for the same object can be active at the same time. The instruction newcog (i.e., "new component object group") creates a new object, but instead of within the current cobox, the new object becomes the *root* of a brand new cobox. It is the root since all other objects which are transitively created using new belong to such new cobox, until other newcog instructions are executed, which introduce other coboxes with their respective roots. We assume all programs include a method called main, which does not belong to any class and has no fields, from which the execution starts in an implicitly created initial cobox, called $\epsilon$.

Program execution is non-deterministic, i.e., given a state there may be different execution steps that can be taken, depending on the cobox selected and, when processors are released, it is also non-deterministic on the particular task within each cobox selected for further execution. We refer to [10] for a precise definition of the language semantics. For our purposes, we only need to know that a program state is formed by a set of coboxes, a set of objects and a set of futures. Each cobox simply contains a unique identifier and the identifier of the currently active object in the cobox (or $\emptyset$ if all objects are idle). Each object contains a unique identifier, the value of its fields, the method name of the active task, and a pool of suspended tasks. Each task in turn contains the values of the local variables and the list of instructions to execute. Execution steps are denoted $S \leadsto_l^b S'$, indicating that we move from state $S$ to state $S'$ by executing instruction $b$ on the object identified by $l$. Traces take the form $t \equiv S_0 \leadsto_{\epsilon}^{b_0} \cdots \leadsto_{l_{n-1}}^{b_{n-1}} S_n$ where $S_0$ is an initial state in which only the main method is available.

*Example 1.* Our running example sketches an implementation of a distributed application to store and retrieve data from a database. The main method creates a new server and initializes it using two arguments, n, the number of *handlers* (i.e., objects that perform requests to the database), and m, the number of requests performed by each handler.

```
void main (Int  n,  Int  m) {
①   Server s = newcog Server(null);
    s!start (n,m);
}
class Server (DAO dao) {
  void start  (Int  n,  Int  m) {
    Fut f <void> = this!initDAO();
    await f ?;
    while(n > 0) {
②     H h = new Handler(this.dao);
      h!run(m);
      n = n − 1;
    }
  }
  void initDAO ()  {
③   this. dao = new DAO(null);
    Fut f <void> = this.dao!initDB();
    await f ?;
  }
}
```

```
class Handler (DAO dao) {
  void run (Int  m) {
    while(m>0) {
      this. dao.query(m);
      m = m − 1;
    }
  }
}
class DAO (DB db) {
  void initDB  ()  {
④   this. db = new DB();
  }
  boolean query(Int m) {
    String  s = . . //query m
    this. db!exec(s);
  }
}
class DB () {
  boolean exec(String s) {. . .}
}
```

Method start initializes a data access object (DAO) that is used by Handler objects to request the database. Then, it creates n Handler objects at program point (p.p. for short) ② and starts their execution via the run method. The DAO object creates a fresh DB object at p.p. ④, that will actually execute queries from handlers. When executing run, each handler performs m requests to the DAO object by invoking method query. The use of Fut<void> variables and await instructions allow method synchronization. Regarding distribution, observe that the configuration contains a single distributed component (the Server cobox at ①), as all other objects are created using new. □

## 3 Background: Points-to and Resource Analysis

In this paper, we make use of the techniques of points-to analysis [11,13] and resource analysis [1,3] to infer quantified abstract configurations. We will try to use them as black boxes along the paper as much as possible. Still, we need to review the basic components that have to be used and/or adapted for our purposes.

### 3.1 Cost Centers and Points-to Analysis

An essential concept of the resource analysis framework for distributed systems in [1,3] is the notion of *cost center*. A cost center represents a distributed component (or node) of the system such that the cost performed on such component can be attributed to its cost center. Since in our language coboxes are the distributed components of the system, finding out the cost centers amounts to inferring the set of coboxes in the program. This can be done by means of points-to analysis [3]. The aim of points-to analysis is to approximate the set of objects (or coboxes) which each reference variable may point to during program execution. Following [11,13], the abstraction of each object created in the program is a syntactic construction of the form $o_{ij\ldots pq}$, where all elements in $ij\ldots pq$ are allocation sites, which represents all run-time objects that were created at $q$ when the enclosing instance method was invoked on an object represented by $o_{ij\ldots p}$, which in turn was created at allocation site $p$. Let $S$ be the set of all allocation sites in a program. Given a constant $k \geq 1$, the analysis considers a finite set of object names, denoted $\mathcal{N}$, which is defined as: $\mathcal{N} = \{\epsilon\} \cup S \cup S^2 \ldots S^k$. Note that $k$ defines the maximum size of sequences of allocations, and it allows controlling the precision of the analysis. Allocation sequences have in principle unbounded length and thus it is sometimes necessary to lose precision during analysis. This is done by just keeping the $k$ rightmost positions in sequences whose length is greater than $k$. We use $|s|$ to denote the length of a sequence $s$. We define the operation $\langle i, j, \ldots, p\rangle \oplus q$ for referring to the following object name: $o_{ij\ldots pq}$ if $|\langle i, j, \ldots, p, q\rangle| \leq k$, or $o_{j\ldots pq}$ otherwise. In addition, a variable can be assigned objects with different object names. In order to represent all possible objects pointed to by a variable, sets of object names are used. We will use the results of the points-to analysis by using $pt(q, x)$ which refers to the set of object names at p.p. $q$ for a given reference variable $x$.

*Example 2.* Let us show (part of) the result of applying the points-to analysis to each program point. Since $\epsilon$ is the first element of all allocation sequences, we omit it.

| | |
|---|---|
| `void start (Int n, Int m) {` | $\{this \mapsto \{o_1\}\}$ |
| `  Fut f<void> = this!initDAO();` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$ |
| `  await f?;` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$ |
| `  while(n > 0) {` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$ |
| ② `H h = new Handler(this.dao);` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$ |
| `    h!run(m);` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$ |
| `    n = n - 1; } }` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$ |
| `void initDAO () {` | $\{this \mapsto \{o_1\}\}$ |
| ③ `this.dao = new DAO(null);` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$ |
| `  Fut f<void> = this.dao!initDB();` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$ |
| `  await f?;` | $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$ |
| `void initDB () {` | $\{this \mapsto \{o_{13}\}\}$ |
| ④ `this.db = new DB();}` | $\{this \mapsto \{o_{13}\}, o_{13}.db \mapsto \{o_{134}\}\}$ |

All object creations use the object name(s) pointed to by this to generate new object names by adding the current allocation site. E.g., at p.p. ②, $this \mapsto \{o_1\}$; the new object name created is $o_{12}$. The set of possible values for this within a method comes from the object name(s) for the variable used to call the method. In what follows, we use $\mathcal{O}$ to refer to the set of object names generated by the points-to analysis. In our example $\mathcal{O}=\{o_\epsilon, o_1, o_{12}, o_{13}, o_{134}\}$. □

## 3.2 Cost Models

Cost models determine the type of resource we are measuring. Traditionally, a cost model $\mathcal{M}$ is a function $\mathcal{M} : Instr \mapsto \mathcal{N}$ which for each instruction in the set of instructions $Instr$ returns a natural number which represents its cost. As an example, if we are interested in counting the number of instructions executed by a program, we define a cost model that counts one unit for any instruction, i.e., $\mathcal{M}(b) = 1$.

In the context of distributed programs, the main difference is that the cost model not only accounts for the cost consumed by the instruction, but it also needs to attribute it to the corresponding cost center. In order to do so, we add an additional parameter to the previous model which corresponds to the allocation site of the cost center: $\mathcal{M}^I(b, o)=c(o)\cdot 1$. As before, we count "1" instruction but now we attribute it to the cost center of the provided object, named $c(o)$. Technically, the way to assign the cost to its corresponding center is by using symbolic cost expressions that contain the cost centers such that, if we are interested in knowing how many instructions have been executed by the cost center $c(o)$, we replace $c(o)$ by 1 and $c(o')$ by 0 for all other $o' {\neq} o$. In the following sections, we will define the cost models that we need for our analysis.

## 3.3 Upper Bounds

Given a set of cost centers $\mathcal{O}$, a definition of cost model $\mathcal{M}$, and a program $P(\overline{x})$, where $\overline{x}$ are the input values for the arguments of the main method, resource analysis obtains an *upper bound* $UB_P^{\mathcal{M}}(\overline{x})$ which is an expression of the form $c(o_1)\cdot e_1 + \ldots + c(o_n)\cdot e_n$ where $o_i \in \mathcal{O}$ and $e_i$ is a cost expression (e.g., polynomials, exponential functions, etc.) with $i = 1, \ldots, n$.

The analysis [1] is object-sensitive in that, given an object $x$ at a program point $p$, it considers the cost for all different possible abstract values in $\mathcal{O}$ that $x$ can take. Technically, this is done by generating cost equations for each possible abstract value and taking the maximum. To allow this object-sensitive extension, the cost model receives the particular allocation site which is being considered by the analysis. The analysis guarantees that $UB_P^{\mathcal{M}}$ is an upper bound on the worst-case cost (for the type of resource defined by $\mathcal{M}$) of the execution of $P$ w.r.t. any input data; and in particular, that each $e_i$ is an upper bound on the execution cost performed within the objects that $o_i$ represents.

Formally, the following theorem states the soundness result of the analysis. Since the length of object names is limited to a length $k$, allocation sequences of length greater than $k$ do not appear as such in the results of points-to analysis. Instead, they are represented by object names which cover them. Therefore, we need some means for relating allocation sequences to the object name which best approximate them. We now define such notion. Given an allocation sequence $l$ and a set of object names $\mathcal{O}$, the *best approximation* of $l$ in $\mathcal{O}$ is the longest object name in $\mathcal{O}$ which covers $l$. I.e., an object name $o_{l'} \in \mathcal{O}$ is the best approximation of $l$ in $\mathcal{O}$ iff $o_l \preceq o_{l'}$ and $\forall o_{l''} \in \mathcal{O} \,.\, o_l \preceq o_{l''} \;\rightarrow\; |l''| < |l'|$ or $l'' = l'$. We use $UB_P^{\mathcal{M}}(\overline{x})|_N$ to denote the result of replacing $c(o_l)$ by 1 if $o_l \in N$ and by 0 otherwise in the resulting UB expression.

**Theorem 1 (soundness [1,3]).** *Let $P$ be a program and $l$ an allocation sequence. Let $\mathcal{O}$ be the object names computed by a points-to analysis of $P$. Let $l'$ be the best approximation of $l$ in $\mathcal{O}$. Then, $\forall \overline{x}, cost(l, P, \overline{x}) \leq UB_P^{\mathcal{M}}(\overline{x}_s)|_{\{o_{l'}\}}$.*

*Example 3.* The UB expression obtained by applying resource analysis on the running example using $\mathcal{M}^I$, which counts the number of instructions executed by each object inferred by the points-to analysis, is $UB_{main}^{\mathcal{M}^I}(n, m) = c(o_1)\cdot 18{+}c(o_{13})\cdot 6{+}c(o_1)\cdot 12\cdot \mathsf{nat}(n){+}$

$c(o_{12}) \cdot 6 \cdot \mathsf{nat}(n) + c(o_{12}) \cdot 8 \cdot \mathsf{nat}(n) \cdot \mathsf{nat}(m) + c(o_{13}) \cdot 2 \cdot \mathsf{nat}(n) \cdot \mathsf{nat}(m) + c(o_{134}) \cdot \mathsf{nat}(n) \cdot \mathsf{nat}(m)$, where $\mathsf{nat}(x) = max(x, 0)$ and it is used for avoiding negative evaluations of cost expressions. In what follows, for readability, nat is omitted from the UB expressions. The number of instructions executed by a particular object name, say $o_{12}$, is obtained as $UB^{\mathcal{M}^I}_{main}(n, m)|_{\{o_{12}\}} = n \cdot (6 + 8 \cdot m)$. Although the resource analysis in [1,3] cannot infer how object identifiers are grouped in the configuration of the program, it can give us the cost executed by a set of objects, $UB^{\mathcal{M}^I}_{main}(n, m)|_{\{o_1, o_{12}\}} = 18 + n \cdot (18 + 8 \cdot m)$.

## 4 Concrete Definitions in Distributed Systems

This section formalizes the concrete notions of configuration and communication that we aim at approximating by static analysis in the next section.

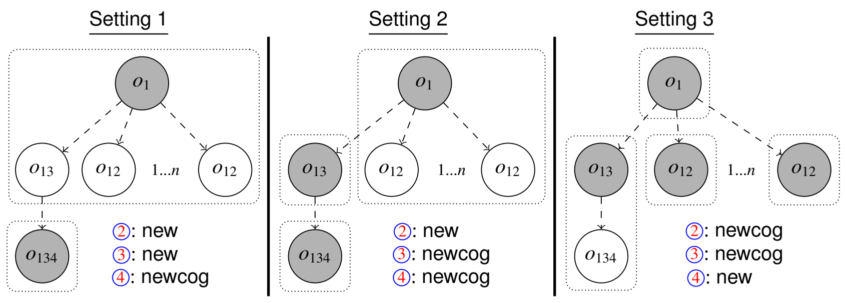
### 4.1 Configuration

Let us introduce some notation. All instructions are labeled. The expression $b \equiv q : i$ denotes that the instruction $b$ has $q$ as label (program point) and $i$ is the instruction proper. Similarly to the points-to analysis defined in Sec. 3.1, any object can be assigned an *allocation sequence* $l = \langle j, \ldots, p, q \rangle$ which indicates that such object was allocated at program point $q$ during the execution of a method invoked on an object whose allocation sequence is in turn $\langle j, \ldots, p \rangle$. We use $o_l$ to refer to an object whose allocation sequence is $l$. Note that allocation sequences are not identifiers since there may be multiple objects with the same allocation sequence. Therefore, we sometimes use multisets (denoted $\{\!\!\{\ \}\!\!\}$). Underscores (_) are used to ignore irrelevant information. Given an allocation sequence $l$, we use $root(l)$ to refer to the allocation sequence of the root object of the cobox for $l$. It can be defined as the longest prefix of $l$ which ends in an allocation site for coboxes, i.e., one site where a newcog instruction is executed. Therefore, if $l$ ends in an allocation site for coboxes, then $root(l) = l$. If it ends in an allocation site for objects, i.e., one where a new instruction is executed, then $root(\langle j, \ldots, p, q \rangle) = root(\langle j, \ldots, p \rangle)$.

Given a trace $t$ (see Section 2), we use $steps(t)$ to denote the set of steps which form trace $t$. Since execution is non-deterministic, given a program $P(\overline{x})$, multiple (possibly infinite) fully expanded traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible fully expanded traces for $P(\overline{x})$. Given a trace $t$, the multiset of cobox roots created during $t$ is defined as $cobox\_roots(t) = \{\!\!\{ o_l \mid \_ \leadsto^{q:\mathsf{newcog}\ \_}_{\langle j,\ldots,p \rangle}\ \_ \in steps(t) \wedge l = \langle j, \ldots, p, q \rangle \}\!\!\}$. Also, given a cobox root $o_l$, the multiset of objects it owns in a trace $t$ is defined as $abs\_in\_cobox(o_l, t) = \{\!\!\{ o_{\langle j,\ldots,p,q \rangle} \mid \_ \leadsto^{q:\mathsf{new}\ \_}_{\langle j,\ldots,p \rangle}\ \_ \in steps(t) \wedge root(\langle j, \ldots, p \rangle) = l \}\!\!\}$.

**Definition 1 (configuration).** *Given an execution trace $t$, we define its* configuration*, denoted $C_t$, as $C_t = \{\!\!\{ \langle o, abs\_in\_cobox(o, t) \rangle \mid o \in cobox\_roots(t) \}\!\!\}$. The* configuration *of a program $P$ on input values $\overline{x}$, denoted $Conf_P(\overline{x})$ is defined as $\{C_t \mid t \in executions(P(\overline{x}))\}$.*

*Example 4.* Deliberately, the running example shown in Ex. 1 executes in a single cobox. It can be configured as a distributed application by creating coboxes instead of objects, i.e., by replacing selected new instructions by newcog . The following graphs graphically show three possible settings and the memory allocation instruction (new or newcog ) that have been used at the program points ②, ③ and ④.



The object names in the graph are grouped using dotted rectangles according to the cobox to which they belong. Cobox roots appear in grey and dashed edges represent the creation sequence. The annotation $1 \ldots n$ indicates that we have $n$ objects of this form. In *Setting 1*, all objects are created in the same cobox, except for the object of type DB. In *Setting 2*, also the object of type DAO is in a separate cobox. In *Setting 3*, each handler is in a separate cobox, and DAO and DB share a cobox. The configurations for the different settings are (see Def. 1):

$$\textit{Setting 1:}\{\!\!\{\langle o_1, \{\!\!\{\underbrace{o_{12},...,o_{12}}_{n\text{ objects}}, o_{13}\}\!\!\}\rangle, \langle o_{134}, \{\!\!\{\}\!\!\}\rangle\}\!\!\} \qquad \textit{Setting 2:}\{\!\!\{\langle o_1, \{\!\!\{\underbrace{o_{12},...,o_{12}}_{n\text{ objects}}\}\!\!\}\rangle, \langle o_{13}, \{\!\!\{\}\!\!\}\rangle, \langle o_{134}, \{\!\!\{\}\!\!\}\rangle\}\!\!\} \quad \square$$

$$\textit{Setting 3:}\{\!\!\{\langle o_1, \{\!\!\{\}\!\!\}\rangle, \underbrace{\langle o_{12}, \{\!\!\{\}\!\!\}\rangle, ..., \langle o_{12}, \{\!\!\{\}\!\!\}\rangle}_{n\text{ coboxes}}, \langle o_{13}, \{\!\!\{o_{134}\}\!\!\}\rangle\}\!\!\}$$

Given input values and an allocation sequence, the definition below counts the maximum number of instances (objects) created at such allocation sequence. We use $card(x, M)$ to refer to the number of occurrences of $x$ in a multiset $M$.

**Definition 2 (number of instances).** *Given an allocation sequence $l$, a program $P$ and input values $\overline{x}$, we define the* number of instances *for $l$ as:*

$$inst(l, P, \overline{x}) = \max_{C_t \in Conf_P(\overline{x})} \Big( \sum_{\langle c, O\rangle \,\in\, C_t} card(l, O \cup \{\!\!\{c\}\!\!\}) \Big)$$

*Example 5.* The number of instances for the allocation sequence $\langle 1, 2\rangle$ in our running example with input values $\overline{x} = \langle 3, 4\rangle$ (i.e., n=3 and m=4) is the maximum number of objects with $\langle 1, 2\rangle$ as allocation sequence, over all possible executions. Such maximum is 3. In fact, for any execution the maximum coincides with the value of n. $\square$

## 4.2 Communication

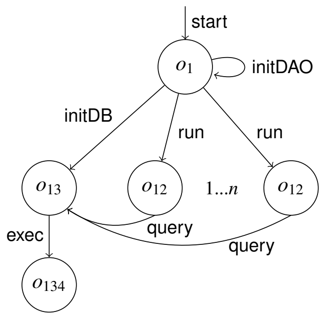
The *communication* refers to the *interactions* between objects occurred during the execution of a program. As in the above section, objects are represented using allocation sequences.

**Definition 3 (communication).** *Given an execution trace $t$, its* interactions*, denoted $I_t$, are defined as: $I_t = \{\!\!\{\langle o_l, o_k, m\rangle \mid \_ \rightsquigarrow_l^{\_ : o_k!m(\_)} \_ \in steps(t)\}\!\!\}$. The* communication *performed in the execution of a program $P$ and input values $\overline{x}$, denoted $Comm_P(\overline{x})$ is defined as $\{I_t \mid t \in executions(P(\overline{x}))\}$.*

A global view of the distributed system for a trace execution $t$ can be depicted as a graph whose nodes are object representations of the form $o_l$, where $l$ is an allocation sequence which occurs in the trace $t$, and whose arcs, annotated with the method name, are given by the elements in the set $I_t$.

*Example 6.* The interactions for any execution of our running example, and thus, the communication of the program, is depicted graphically in the following graph and, according to Def. 3 it is defined as:

$$\{\!\!\{\langle \epsilon, o_1, \mathsf{start}\rangle, \langle o_1, o_1, \mathsf{initDAO}\rangle, \langle o_1, o_{13}, \mathsf{initDB}\rangle, \underbrace{\langle o_1, o_{12}, \mathsf{run}\rangle, ..., \langle o_1, o_{12}, \mathsf{run}\rangle}_{n \text{ interactions}}, \underbrace{\langle o_{12}, o_{13}, \mathsf{query}\rangle, ..., \langle o_{12}, o_{13}, \mathsf{query}\rangle}_{n{\cdot}m \text{ interactions}} \underbrace{\langle o_{13}, o_{134}, \mathsf{exec}\rangle, ..., \langle o_{13}, o_{134}, \mathsf{exec}\rangle}_{n{\cdot}m \text{ interactions}} \}\!\!\}$$



Observe that the communication of the program comprises all calls to methods, including calls within the same object such as $\langle o_1, o_1, \mathsf{initDAO}\rangle$. A relevant aspect of communications is that they are independent from the distributed setting of the program. □

**Definition 4 (number of interactions).** *Given two allocation sequences $l$ and $k$, a method $m$, a program $P$ and its input values $\overline{x}$, we define the* number of interactions *between $l$ and $k$ for method $m$ in the execution of $P$ on $\overline{x}$ as:* $ninter(l, k, m, P, \overline{x}) = \max\limits_{I_t \in Comm_P(\overline{x})} (card(\langle l, k, m\rangle, I_t))$.

*Example 7.* In the running example, methods initDAO and initDB are executed only once. During the execution of start in object $o_1$, method run is called inside the while loop and it is executed $n$ times by the objects $o_{12}$. Similarly, for each execution of run in $o_{12}$, method query is called $m$ times, resulting in $n \cdot m$ calls to method query in $o_{13}$. Besides, each call to query executes exec in $o_{134}$. □

## 5 Inference of Quantified Abstractions

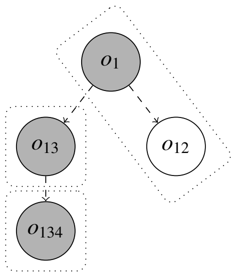
This section presents our method to infer quantified abstractions of distributed systems. The main novelties are: (1) We provide an abstract definition for configuration and communication that can be automatically inferred by relying on the results computed by points-to analysis. (2) We enrich the abstraction by integrating quantitative information inferred by resource analysis. For this, we build on prior work on resource analysis [1,3] that was primarily used for the estimation of upper bounds on the worst-case cost performed by each node in the system (see Section 3). To use this analysis, we need to define new cost models that allow establishing upper bounds for the number of nodes and communications which the execution of the system requires.

### 5.1 Quantified Configurations

The points-to analysis results can be presented by means of a *points-to graph* as follows. We use $alloc(P)$ to denote the set of allocation sites in program $P$.

**Definition 5 (points-to graph).** *Given a program $P$ and its points-to analysis results, we define its* points-to graph *as a directed graph $G_P = \langle V, E\rangle$ whose set of nodes is $V = O$ and set of edges is $E = \{o_l \rightarrow o_{l'} \mid q{:}y{=}\mathsf{new}\ \_$ or $q{:}y{=}\mathsf{newcog}\ \_ \in alloc(P) \wedge o_l \in pt(q, this) \wedge o_{l'} \in pt(q, y)\}$.*

*Example 8.* The following graph shows the points-to graph for the running example. It contains one node for each object name inferred by the points-to analysis. Given an allocation site, edges link object names pointed to by this to the corresponding objects created at that program point, e.g., an edge from $o_1$ to $o_{13}$ and $o_{12}$ and another one from $o_{13}$ to $o_{134}$. □



Points-to graphs provide abstractions of the ownership relations among objects in the program. To extract abstract configurations from them, it is necessary to identify cobox roots and find the set of objects which belong to the coboxes associated to such roots. Note that given an object name $\langle j \dots q\rangle$ it can be decided whether it represents a cobox root, denoted $is\_root(\langle j \dots q\rangle)$, by simply checking whether the allocation site $q$ contains a newcog instruction. We write $a \rightsquigarrow b$ to indicate that there is a non-empty path in a graph from $a$ to $b$ and denote by $interm(a, b)$ the set of intermediate nodes in the path (excluding $a$ and $b$).

**Definition 6 (abstract configuration).** *Given a program $P$ and a points-to graph $G_P = \langle V, E\rangle$ for $P$, we define its* abstract configuration *$\mathcal{A}_P$ as the set of pairs of the form $\langle o, abs\_in\_cobox(o, G_P)\rangle$ s.t. $o \in V \wedge is\_root(o)$ where $abs\_in\_cobox(o, G_P) = \{o' \in V$ s.t. $o \rightsquigarrow o'$ in $G_P$ and $\forall o'' \in interm(o, o') \ \wedge\ \neg is\_root(o'')\}$.*

Note that, in the above definition, function *abs_in_cobox* returns the subset of objects which are part of the cobox whose root is the parameter $o$.

*Example 9 (abstract configuration).* The abstract configuration for the concrete *Setting 2* is represented graphically in Ex. 8. As before, cobox roots appear in grey and objects are grouped by cobox. The abstract configurations for the Settings in Ex. 4 are:
*Setting 1*: $\langle o_1, \{o_{12}, o_{13}\}\rangle, \langle o_{134}, \{\}\rangle$, *Setting 2*: $\langle o_1, \{o_{12}\}\rangle, \langle o_{13}, \{\}\rangle, \langle o_{134}, \{\}\rangle$,
*Setting 3*: $\langle o_1, \{\}\rangle, \langle o_{12}, \{\}\rangle, \langle o_{13}, \{o_{134}\}\rangle$ □

Soundness of the analysis requires that the abstract configuration obtained is a safe approximation of the configuration of the program for any input values. Given two object names $o_l$ and $o_{l'}$, we say that *$o_{l'}$ covers $o_l$*, written $o_l \leq o_{l'}$ if $l'$ is a suffix of $l$ modulo $\oplus$. Given two sets of object names $O_1$ and $O_2$, we write $O_1 \sqsubseteq O_2$ if all objects in $O_1$ are covered by some element in $O_2$. Given $\langle o_l, O\rangle$ and $\langle o_{l'}, O'\rangle$, we write $\langle o_l, O\rangle \sqsubseteq \langle o_{l'}, O'\rangle$ if $o_l \leq o_{l'}$ and $O \sqsubseteq O'$. Given two configurations $C$ and $C'$, we write $C \sqsubseteq C'$ if $\forall \langle o_l, O\rangle \in C$ there exists $\langle o_{l'}, O'\rangle \in C'$ s.t. $\langle o_l, O\rangle \sqsubseteq \langle o_{l'}, O'\rangle$.

**Theorem 2 (soundness of abstract configurations).** *Let $P$ be a program and $\mathcal{A}_p$ its abstract configuration. Then $\forall \overline{x}, \forall C_t \in Conf_P(\overline{x}), C_t \sqsubseteq \mathcal{A}_p$.*

The proof is entailed from the soundness proof of the underlying points-to analysis (our implementation uses an adaptation of [11]). It is easy to see that the theorem holds for the configuration $Conf_P$ in Ex. 4, and any abstract configuration $\mathcal{A}_P$ of Ex. 9.

(Non-quantified) abstract configurations are already useful when combined with the resource analysis in Sec. 3, since they allow us to obtain the resource consumption at the level of cobox names. In what follows, given a points-to graph $G_P$ and a cobox root $o$, we use $cobox(o, G_P,)$ to denote $\{o\} \cup abs\_in\_cobox(o, G_P)$.

*Example 10.* Using the UB expression inferred in Ex. 3 and the abstract configurations for all settings in Ex. 9, we can obtain the cost for each cobox name. The following table shows the results obtained from $UB^{\mathcal{M}^I}_{main}(n, m)|_{cobox(c,G_{main})}$ where $c$ corresponds, in each case, to the cobox name in the considered abstract configuration:

| *Setting 1* | | *Setting 2* | | *Setting 3* | |
|---|---|---|---|---|---|
| $c$ | $UB$ | $c$ | $UB$ | $c$ | $UB$ |
| $o_1$ | $24 + 18{\cdot}n + 10{\cdot}n{\cdot}m$ | $o_1$ | $18 + 18{\cdot}n + 8{\cdot}n{\cdot}m$ | $o_1$ | $18 + 12{\cdot}n$ |
| $o_{134}$ | $n{\cdot}m$ | $o_{13}$ | $6 + 2{\cdot}n{\cdot}m$ | $o_{12}$ | $6{\cdot}n + 8{\cdot}n{\cdot}m$ |
| | | $o_{134}$ | $n{\cdot}m$ | $o_{13}$ | $6 + 3{\cdot}n{\cdot}m$ |

As the table shows, in *Settings 1* and *2* most of the instructions are executed in cobox(es) represented by cobox name $o_1$. In *Setting 3*, the cost is more evenly distributed among cobox names. However, in order to reason about how loaded actual coboxes are it is required to have information about how many instances of each cobox name exist. For example, in *Setting 3*, $o_{12}$ represents $n$ Handler coboxes. This essential (and complementary) information will be provided by the quantified abstraction. □

We now aim at quantifying abstract configurations, i.e., at inferring an over-approximation of the number of concrete objects (and coboxes) that each abstract object (or cobox) represents. For this purpose, we define the $\mathcal{M}^C(b, o_l)$ cost model as a function which returns $c(o_{l \oplus q})$ if $b \equiv q{:}y{=}\text{new}\ C$ or $b \equiv q{:}y{=}\text{newcog}\ C$, and 0 otherwise. The novelty is on how the information computed by the points-to analysis is used in the cost model: it concatenates the allocation sequence of the object received as parameter (that corresponds to the considered allocation sequence for this) with the instruction allocation site $q$. This allows counting the elements created at this point for each particular instance of this considered by the points-to analysis.

*Example 11.* Using $\mathcal{M}^C$, the upper bound obtained for the running example is the expression $UB^{\mathcal{M}^C}_{main}(n, m) = c(o_1) + c(o_{13}) + c(o_{134}) + n \cdot c(o_{12})$. This expression allows us to infer an upper bound of the maximum number of instances for any object identified in the points-to graph. Regarding configurations, we are interested in the number of instances of those objects that are distributed nodes (coboxes). The following table shows the results of solving the expression $UB^{\mathcal{M}^C}_{main}(n, m)|_{cobox(c,G_{main})}$ where $c$ as before are the coboxes for each abstract configuration.

| *Setting 1* | | *Setting 2* | | *Setting 3* | |
|---|---|---|---|---|---|
| $c$ | $UB$ | $c$ | $UB$ | $c$ | $UB$ |
| $o_1$ | 1 | $o_1$ | 1 | $o_1$ | 1 |
| $o_{134}$ | 1 | $o_{13}$ | 1 | $o_{12}$ | n |
| | | $o_{134}$ | 1 | $o_{13}$ | 1 |
| 2 | | 3 | | $2 + n$ | |

Clearly, *Setting 1* is the setting that creates fewer coboxes (only 2 coboxes execute the whole program). Thus, the queries requested by handlers cannot be processed in parallel. If there is more parallel capacity available, *Setting 3* may be more appropriate, since handlers can process requests in parallel.

**Theorem 3.** *Under the assumptions in Th. 1,* $\forall \overline{x}$, $inst(l, P, \overline{x}) \leq UB^{\mathcal{M}^C}_P(\overline{x}_s)|_{\{o_{l'}\}}$.

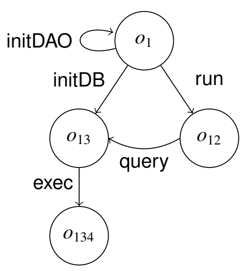
The proof is an instance of Th. 1 for $\mathcal{M}^C$ and the definition of *inst* in Def. 2.

## 5.2 Quantified Communication

From the points-to analysis results, we can generate the *interaction graph* as follows.

**Definition 7 (interaction graph).** *Given a program $P$ and its points-to analysis results, we define its* interaction graph *as a directed graph $I_P = \langle V, E\rangle$ with a set of nodes $V = O$ and a set of edges $E = \{o_l \xrightarrow{m} o_{l'} \mid q{:}x!m(\_) \wedge o_l \in pt(q, this) \wedge o_{l'} \in pt(q, x)\}$.*

*Example 12.* The following graph shows the interaction graph for the example. Edges connect the object that is executing when a method is called with the object responsible for executing the call, e.g., during the execution of start, object $o_1$ calls method initDAO using the this reference and it also interacts with $o_{12}$ by calling run. Note that the multiple calls to query from $o_{12}$ to $o_{13}$ are abstracted by one edge. □



We now integrate quantitative information in the interaction graph. For this purpose, we define the cost model $\mathcal{M}^K(b, o, p)$ as a function which returns $c(m){\cdot}c(o, p)$ if $b \equiv \_!m(\_)$, and 0 otherwise. The key point is that for capturing interactions between objects, when applying the cost model to an instruction, we pass as parameters the considered allocation sequences of the caller and callee objects. The resulting upper bounds will contain cost centers made up of pairs of abstractions $c(o, p)$, where $o$ is the object that is executing and $p$ is the object responsible for executing the call. Besides, we attach to the interaction the name of the invoked method $c(m)$ (multiplication is used as an instrument to attach this information and manipulate it afterwards as we describe below).

From the upper bounds on the interactions, we can obtain a range of useful information: (1) By replacing $c(m)$ by 1, we obtain an upper bound on the number of interactions between each pair of objects. (2) We can replace $c(m)$ by (an estimation of) the amount of data transferred when invoking $m$ (i.e., the size of its arguments). This is a first approximation of a band-width analysis. (3) Replacing $c(o, p)$ by 1 for selected objects and the remaining ones by 0, we can see the interactions between the selected objects. (4) If we are interested in the communications for the whole program, we just replace all expressions $c(o, p)$ by 1. (5) Furthermore, we can obtain the interactions between the distributed nodes by replacing by 1 those cost centers in which $o$ and $p$ belong to different coboxes and by 0 the remaining ones. From this information, we can detect nodes that have many interactions and that would benefit from being deployed on the same machine or at least have a fast communication channel.

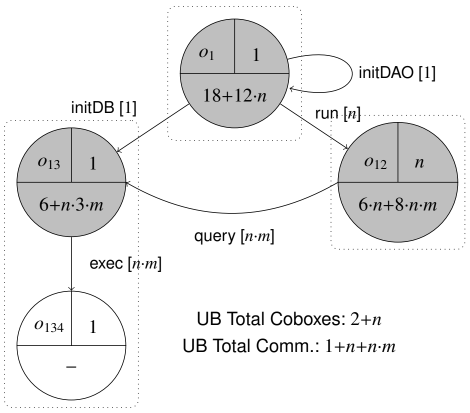
*Example 13.* The interaction UB obtained by the resource analysis is as follows:

$$UB^{\mathcal{M}^K}_{main}(n, m) = c(\mathsf{start}){\cdot}c(\epsilon, o_1) + c(\mathsf{initDAO}){\cdot}c(o_1, o_1) + c(\mathsf{initDB}){\cdot}c(o_1, o_{13}) + \\ n{\cdot}c(\mathsf{run}){\cdot}c(o_1, o_{12}) + n{\cdot}m{\cdot}c(\mathsf{query}){\cdot}c(o_{12}, o_{13}) + n{\cdot}m{\cdot}c(\mathsf{exec}){\cdot}c(o_{13}, o_{134})$$

From this global UB, we obtain the following UBs on the number of interactions between coboxes for the different settings in Ex. 4:

| *Setting 1* | | | *Setting 2* | | | *Setting 3* | | |
|---|---|---|---|---|---|---|---|---|
| *method* | *coboxes* | *UB* | *method* | *coboxes* | *UB* | *method* | *coboxes* | *UB* |
| exec | $o_1 \rightarrow o_{134}$ | $n{\cdot}m$ | query | $o_1 \rightarrow o_{13}$ | $n{\cdot}m$ | run | $o_1 \rightarrow o_{12}$ | n |
| | | | exec | $o_{13} \rightarrow o_{134}$ | $n{\cdot}m$ | initDB | $o_1 \rightarrow o_{13}$ | 1 |
| | | | | | | query | $o_{12} \rightarrow o_{13}$ | $n{\cdot}m$ |
| $n{\cdot}m$ | | | $n{\cdot}m + n{\cdot}m$ | | | $1 + n + n{\cdot}m$ | | |

The last row shows the total number of interactions between coboxes. Clearly, the minimum number of inter-cobox interactions happens in *Setting 1*, where most of the objects are in the same cobox. *Setting 2* has a higher number of interactions, because the database objects DAO and DB are in different coboxes. In *Setting 3* most interactions are produced between the coboxes created for the handlers which, on the positive side, may run in parallel. By combining this information with the quantified configuration of the system, for *setting 3*, we generate the *quantified abstraction* (shown in the graph). Each node contains as object identifier its allocation sequence and the number of instances (e.g., the number of instances of $o_{12}$ is $n$). Optionally, if it is a cobox, it contains the number of instructions executed by it. For instance, the UB on the number of instructions executed in cobox $o_{12}$ is $6{\cdot}n{+}8{\cdot}n{\cdot}m$ (see Ex. 10).



The edges represent the interactions and are annotated (in brackets) with the UB on the number of calls (e.g., the objects represented by $o_{12}$ call to $o_{13}$ $n{\cdot}m$ times calling method query). ◻

From the example, we can figure out the five applications described in Sec. 1: (1) We can visualize the topology and view the number of tasks to be executed by the distributed nodes and possibly spot errors. (2) We detect that node $o_1$ executes only one process, while $o_{13}$ executes many. Thus, it probably makes sense to have them sharing the processor. (3) We can perform meaningful resource analysis by assigning to each distributed node the number of steps performed by it, rather than giving this number at the level of objects as in [1,3] (as maybe the objects do not share the processor). (4) We can see that $o_{13}$ and $o_{12}$ have many interactions and would benefit from having a fast communication channel. (5) From the quantified interactions, if we compute the sizes of the arguments, we can figure out the size of the data to be transferred (bandwidth

Similarly to abstract configurations, we use $UB^{\mathcal{M}^K}_{main}(\overline{x})|_{N,M}$ to denote the result of replacing in the resulting UB the expression: $c(o_1, o_2)$ by 1 if $o_1, o_2 \in N$ and by 0 otherwise and $c(m)$ by 1 if $m \in M$ and by 0 otherwise. This theorem is also an instance of Th. 1 for the defined cost model and Def. 4.

**Theorem 4 (soundness).** *Under the assumptions in Theorem 1,* $\forall\ \overline{x}$ *we have that* $ninter(l, k, m, P, \overline{x}) \leq UB^{\mathcal{M}^K}_{P}(\overline{x})|_{\{o_l, o_k\},\{m\}}$.

## 6 Implementation and Application to Case-Study

We have implemented our analysis in COSTABS [1] and applied it to a realistic case-study, the Trading System developed by Fredhopper® and available from `http://www.hats-project.eu`. Due to some limitations of the underlying resource analysis which are not related to our method, we had to slightly modify the program by changing the structure of some loops, and had to add some size relations that the analysis could not infer. The simple online interface to our analysis and the modified case-study can be found at `http://costa.ls.fi.upm.es/costabs/QA`. This Trading System is a typical example for a distributed component-based information system. It models a supermarket sales handling system: it includes the processes at a single cash desk (like scanning products using a bar code scanner or paying by cash or by credit card); it also handles bill printing; as well as administrative tasks In the Trading System, a store consists of an arbitrary number of cash desks. Each of them is connected to the store server, holding store-local product data such as inventory stock, prices, etc.

The experiments presented have been performed on an Intel Core 2 Duo at 2.53GHz with 4GB of RAM, running Ubuntu 12.10. The analyzed source code has 1340 l.o.c., with 94 methods and 22 classes. Points-to analysis has been performed with $k = 2$, i.e., the maximum length of object names (see Sec. 3) is two. The inference of the quantified configuration took 109 seconds and of the quantified communication 410 seconds. The larger time taken by the communication is justified because there are many more method invocations than object creations in the program and, thus, the resource analysis has more equations to solve in the latter case. The analysis identifies 22 different object names (17 of them are coboxes) and 96 interactions in the communication graph. We do not show the UBs inferred because the expressions obtained are rather large.

The QA obtained for the system is as follows: we identify two separate parts in the model, an environment part that creates a handler for each physical device and another part that represents the physical devices. The configuration of the system is distributed by creating one cobox for each physical device. The quantified abstraction infers that the number of instances for each identified cobox is linear on the number of cash desks installed. Besides, the system also creates one distributed environment object running on its own cobox for handling each physical device. The interactions of the system show that each environment cobox communicates with its physical device in order to perform each task. A notable result of our experiments is that we have detected two objects with a high number of interactions, namely *CashDeskPCImpl* and *CashBox-Impl*, and which run in separate coboxes. Clearly, the implementation would benefit from deploying these two coboxes on the same machine since their tasks are highly cooperative. If this is not possible, it should be at least guaranteed that they have a fast communication channel. The remaining objects do not show any overloading problem.

## 7 Conclusions and Future Work

We have shown that distributed systems can be statically approximated, both qualitatively and quantitatively. For this, we have proposed the use of powerful techniques for points-to and resource analysis whose integration results in a novel approach to describing system configurations. There exist several contributions in the literature about occurrence counting analysis in mobile systems of processes, although they focus on high-level models, such as the $\pi$-calculus and BioAmbients [8,9]. But, to the best of our knowledge, this paper is the first approach that presents a quantitative abstraction of a distributed system for a real language and experimentally evaluates on a prototype. We argue that our work is a first crucial step towards automatically inferring optimal deployment configurations of distributed systems. In future work, we plan to tackle this problem and consider objective functions. An objective function should indicate the cost metrics that we aim at keeping minimal, e.g., by taking into account the actual features of the deployment platforms.

*Acknolegments:* This work was funded partially by EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and by the Madrid Regional Government project S2009TIC-1465.

## References

1. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *APLAS'11*, vol. 7078 of *LNCS*, p.p. 238–254. Springer, 2011.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *Procs. of PEPM'12*, p.p. 151–154. ACM Press, 2012.
3. E. Albert, P. Arenas, J. Correas, M. Gómez-Zamalloa, S. Genaim, G. Puebla, and G. Román-Díez. Object-sensitive cost analysis for concurrect objects. *Technical Report*, http://costa.ls.fi.upm.es/papers/costa/AlbertACGGPRtr.pdf, 2012.
4. P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.
5. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
6. D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
7. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM Press, 1978.
8. J. Feret. Occurrence counting analysis for the pi-calculus. *ENTCS*, 39(2):1–18, 2001.
9. R. Gori and F. Levi. A new occurrence counting analysis for bioambients. In *APLAS'05*, vol. 3780 of *LNCS*, p.p. 381–400. Springer, 2005.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO*, vol.6957 *LNCS*, p.p.142–164. Springer, 2010.
11. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.
12. J. Schäfer and A. Poetzsch-Heffter. JCobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, LNCS 6183, p.p. 275–299. Springer, 2010.
13. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *Procs. of POPL'11*, p.p. 17–30. ACM, 2011.
14. A. Yonezawa, J.P. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. In *Procs. of OOPLSA'86*, p.p. 258–268, USA. ACM, 1986.