Under consideration for publication in Formal Aspects of Computing

Quantified Abstract Configurations of Distributed Systems**

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

[1] DSIC, Complutense University of Madrid
[2] DLSIIS, Technical University of Madrid

Abstract. When reasoning about distributed systems, it is essential to have information about the different kinds of nodes that 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 *resource analysis* that limit the number of concrete instances that the nodes and arcs represent and their resource consumption. The information available in our *quantified abstract configurations* allows us to define performance indicators which measure the quality of the system. In particular, we present several indicators that assess the level of distribution in the system, the amount of communication among distributed nodes that it requires, and how balanced the load of the distributed nodes that compose the system is. Our performance indicators are given as functions on the input data sizes, and they can be used to automate the comparison of different distributed settings and guide towards finding the optimal configuration.

Keywords: Static Analysis; Cost Analysis; 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 that 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 [BYV09]

Correspondence and offprint requests to: Elvira Albert, email: elvira@sip.ucm.es

** **Author's version.** The final publication is available at http://link.springer.com/article/10.1007/s00165-014-0321-z is an example of this). In this paper, we introduce the notion of *Quantified Abstraction* (QA for short) of a distributed system that 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 that each abstract node represents, and an upper bound on the (possibly infinite) number of actual interactions that each abstract interaction represents. 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.

We apply our analysis to an Actor-like language [JHS12] for distributed concurrent systems based on asynchronous communication. Actors form a well established model for distributed systems [YBS86, Ame89, Car93, SPH10], which is lately regarding attention due to its adoption in Erlang [AVW96], Scala [HO09] and active objects [SPH10, BGS12]. In our language, the distribution model is based on (possibly interacting) objects that 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 that 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 (or interactions) between each pair of objects in the system; from which we can later obtain the external 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. In our approach, QAs are inferred in two main steps; first, we infer the nodes that safely describe all possible executions that might occur at runtime, and, second, we infer the interactions between the nodes identified for the configurations. An allocation site is a program point in which an object is created. The abstraction of objects and coboxes we rely on is based on *allocation sequences* [MRR05]. An allocation sequence is the sequence of allocation sites where the objects that led to the creation of the current one were created. By using this abstraction, a quantified abstraction of a distributed system is a directed graph whose *nodes* represent the abstract objects that may be created along the execution, and the *edges* link a node n_1 with a node n_2 to represent a call (or interaction) from an object in n_1 to an object in n_2. QAs are quantified since nodes and edges are enriched with upper bounds. In the case of nodes, the upper bound (over)approximates the number of concrete instances of each abstract object. For edges, the upper bound (over)approximates the number of calls among the objects.

The main applications of QAs are on optimizing, debugging and efficiently dimensioning distributed systems because: (1) QAs provide a global view of the distributed application, which may help to find the best task distribution, 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. This allows us to check if the configuration is well balanced. (4) They allow us to detect components that have many communications and that would benefit from being deployed in the same machine or at least have a very fast communication channel.

1.1. Summary of contributions

The main contributions of this paper can be summarized as follows:

1. *Abstract configurations.* We use a points-to analysis to infer the allocation sequences for reference variables. Such sequences allow us to infer the ownership relations that determine to which coboxes the objects belong. From this information, we compute an abstraction of the configuration of the distributed system.
2. *Quantified nodes.* We define a new cost model that can be plugged in the generic resource analyzer SACO

[AAG12a] (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.* Likewise, we propose a cost model that can be also plugged in SACO to infer upper bounds on the number of calls among nodes. The three points described so far allow us to define QAs.
4. *Performance indicators.* We define performance indicators that can be obtained from the information gathered in the QAs and that allow us to evaluate the quality of a particular distributed system.
5. *Optimal settings.* Performance indicators can be used together with deployment constraints (i.e., restrictions given by the concrete deployment scenario) to find the configuration that best satisfies the constraints imposed. For this purpose, we outline practical ways to determine when one distributed setting is *better* than another one.
6. *Implementation.* We have implemented our analysis in SACO (`http://costa.ls.fi.upm.es/SACO`) and applied it on several benchmarks and on two larger case studies. Our experiments show that the application of our approach to finding optimal configurations is feasible and useful to guide the deployment of a distributed program.

This article improves and extends the iFM'13 Conference paper [ACP13] in the following aspects: (1) it improves the formalization by giving the semantics of the language and proving the soundness of our approach w.r.t. such semantics, (2) we introduce several performance indicators that can be automatically inferred using our techniques, (3) we develop the application of QAs to find optimal configurations of distributed systems that adhere to given deployment constraints and (4) we extend our experimental evaluation by applying our techniques for finding optimal configurations to some classical distributed applications and two case studies.

1.2. Organization of the article

The article is organized as follows. Section 2 describes the *a*bstract *b*ehavioral *s*pecification language (ABS) and its semantics. This is a language for designing distributed object-oriented systems that uses the concept of coboxes that execute concurrently. When an ABS program is written, the programmer may decide whether an object is created in the current cobox or in a fresh cobox by using different language constructs for object creation, namely `new` and `newcog`, respectively. Different distributed *settings* are thus achieved by trying different combinations of `new` and `newcog` instructions.

Background concepts needed throughout the article are introduced in Section 3. First, the points-to analysis is briefly described. It is used to obtain information about the coboxes created by the program and how objects are distributed among them. The result of the points-to analysis is later used to define *cost centers* for which our static analysis will compute quantitative information about a resource being measured. The resource of interest is specified by the definition of a *cost model*, a function that relates each type of instruction with a number that represents its cost. Given a cost model, the resource analysis obtains an *upper bound* of the cost of executing a program.

Section 4 defines the concrete notions of *configuration* and *communication* for a given program execution. The configuration collects the state of a program regarding objects and coboxes for all possible execution traces. The communication refers to the interactions between objects during the execution of a program. A method for inferring an over-approximation of both concepts is described in Section 5, by means of an abstract definition of configuration and communication, and by integrating quantitative information based on points-to and resource analysis. Soundness of the analysis results is proved.

The information available in our *quantified abstract configurations* is used in Section 6 to define a series of performance indicators that measure the quality of a setting and allow us to compare it to other settings. In particular, we present several indicators that assess the level of distribution in the setting, the amount of communication with other distributed nodes that it requires, and how balanced the load of the distributed nodes that compose the system is. Since software performance can vary significantly depending on the target architecture, *deployment constraints* can be used to express decisions that reflect the deployment scenario.

Section 7 describes the experimental evaluation of our approach for several classical distributed applications and how our techniques can be used for finding the optimal configuration of a system. Section 8 overviews other approaches in the literature and relates them to our work. Finally, Section 9 summarizes the main conclusions of this article and points out several directions for future research.

2. The Language

We apply our analysis to the language ABS (Abstract Behavioural Specification language) [JHS12, SPH10]. ABS extends the basic concurrent objects model [YBS86, Ame89, Car93, SPH10] 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 $y = o!m()$. The objects *this* and o communicate by means of the invocation m. Here, y is a future variable that allows synchronizing with the completion of task m by means of the await $y?$ instruction that 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

$$\texttt{class}\ C_1\ \texttt{(}t_1\ fn_1\texttt{,}...\texttt{,}t_n\ fn_n\texttt{)}\ \texttt{\{}M_1\ \ldots\ M_k\texttt{\}}$$

where each $t_i\ fn_i$ declares a field fn_i of type t_i, and each M_i is a *method definition*

$$t\ \ m\texttt{(}t_1\ \ w_1\texttt{,}...\texttt{,}t_n\ \ w_n\texttt{)}\ \ \texttt{\{}t_{n+1}\ \ w_{n+1}\texttt{;}...\texttt{;}t_{n+p}\ \ w_{n+p}\texttt{;}\ \ s\texttt{\}}$$

where t is the type of the return value; $w_1, ..., w_n$ are the formal parameters with types $t_1, ..., t_n$; $w_{n+1}, ..., w_{n+p}$ are local variables with types $t_{n+1}, ..., t_{n+p}$; and s is a sequence of instructions that adheres to the grammar below, where we use x and z to denote standard variables, y to denote a future variable whose declaration includes the type of the returned value, $\overline{x}$ to denote a sequence of variables of the form $x_1, \ldots x_n$, and f for representing a field. For the sake of generality, the syntax of expressions e and types t is left open.

$$\begin{array}{rcl} s & ::= & in \mid in; s \\ in & ::= & x = \mathsf{new}\ C(\overline{x}) \mid x = \mathsf{newcog}\ C(\overline{x}) \mid x = e \mid \mathsf{this}.f = e \mid y = x!m(\overline{z}) \mid y = \mathsf{this}!m(\overline{z}) \mid \\ & & y = \mathsf{this}.f!m(\overline{z}) \mid \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{return}\ x \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{await}\ y? \end{array}$$

There is an implicit local variable called this that refers to the current object. Observe that the only fields that can be accessed are those of the current object, i.e., this. Thus, the language is data-race free [JHS12], 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, which becomes the *root* of a brand new cobox. It is the root since all other objects that are transitively created using new belong to such new cobox, until another newcog instruction is executed, which will introduce another cobox. 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 c_ϵ.

Figure 1 presents the semantics of the language for the instructions related to concurrency. The instructions for sequential execution (i.e., assignment, condition and iteration) are standard and thus omitted. 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, also, when processors are released, it is non-deterministic on the particular task within each cobox selected for further execution. A program state is formed by a set of coboxes $cobox(c, o)$, objects $ob(o, a, t, t_s)$ and futures $fut(f, v)$. Each cobox simply contains a unique identifier c and the identifier of the currently active object in the cobox o (or *idle* if all objects in the cobox are idle). Each object contains a unique identifier o, a mapping a from the object fields to their values, the active task t, and a pool of pending tasks t_s. Each task in turn is a pair $\langle tv, s\rangle$ that contains a mapping from the local variables tv to their values, and the list of instructions s to execute. The set of fields of an object contains a special field named *cobox* with a reference to the cobox where the object is executing. Analogously, the set of local variables of a given method contains a special variable *ret* to store the future variable that will receive the result of the method. Futures $fut(f, v)$ in the program state are used to synchronize the execution of a task with the termination of another one, and return the result of executing an asynchronous call, as we explain in the rules below.

The following auxiliary functions are used in the semantics. Function $fresh(n)$ returns a globally unique identifier, which can be used for referring to an object, a future variable or a cobox. Function $initAtts(C, \bar{v}, c)$ is used for field initialization, it returns the initial state of an instance of class C, with formal parameters bound to $\bar{v}$ and *cobox* bound to c. Function $buildLocals(\bar{w}, o, f, m)$ produces the initial values of local variables for method m from the actual parameters $\bar{w}$. The name f is a fresh reference to the future $fut(f, v)$ in the program state, and it is used as value for the special local variable *ret*. This allows notifying the caller the

$$(\textsf{new-obj})\ \frac{fresh(o') \quad a' = initAtts(C, tv(\bar{z}), c)}{ob(o, a, \langle tv, (x = \textsf{new}\ C(\bar{z}); s)\rangle, q)\ cobox(c, o) \rightsquigarrow ob(o, a, \langle tv[x \mapsto o'], s\rangle, q)\ cobox(c, o)\ ob(o', a', idle, \emptyset)}$$

$$(\textsf{new-cobox})\ \frac{fresh(o') \quad fresh(c') \quad a' = initAtts(C, tv(\bar{z}), c')}{ob(o, a, \langle tv, (x = \textsf{newcog}\ C(\bar{z}); s)\rangle, q) \rightsquigarrow ob(o, a, \langle tv[x \mapsto o'], s\rangle, q)\ ob(o', a', idle, \emptyset)\ cobox(c', idle)}$$

$$(\textsf{async-call})\ \frac{o' = tv(x) \quad fresh(f) \quad tv' = buildLocals(tv(\bar{z}), o', f, m)}{\begin{array}{c} ob(o, a, \langle tv, (y = x!m(\bar{z}); s)\rangle, q)\ ob(o', a', p', q') \rightsquigarrow \\ ob(o, a, \langle tv[y \mapsto f], s\rangle, q)\ ob(o', a', p', q' \cup \{\langle tv', body(m)\rangle\})\ fut(f, \bot) \end{array}}$$

$$(\textsf{return})\ \frac{v = tv(x) \quad f = tv(ret) \quad c = a(cobox)}{ob(o, a, \langle tv, (\textsf{return}\ x; s)\rangle, q)\ cobox(c, o)\ fut(f, \bot) \rightsquigarrow ob(o, a, idle, q)\ cobox(c, idle)\ fut(f, v)}$$

$$(\textsf{await-t})\ \frac{tv(y) = f \quad v \neq \bot}{ob(o, a, \langle tv, (\textsf{await}\ y?; s)\rangle, q)\ fut(f, v) \rightsquigarrow ob(o, a, \langle tv[y \mapsto v], s\rangle, q)}$$

$$(\textsf{await-f})\ \frac{tv(y) = f \quad c = a(cobox)}{ob(o, a, \langle tv, (\textsf{await}\ y?; s)\rangle, q)\ cobox(c, o)\ fut(f, \bot) \rightsquigarrow ob(o, a, idle, q \cup \{\langle tv, (\textsf{await}\ y?; s)\rangle\})\ cobox(c, idle)\ fut(f, \bot)}$$

$$(\textsf{activate})\ \frac{q \neq \emptyset \quad o = selectObj(S) \quad p = selectTask(q) \quad c = a(cobox)}{ob(o, a, idle, q)\ cobox(c, idle) \rightsquigarrow ob(o, a, p, q \setminus \{p\})\ cobox(c, o)}$$

Fig. 1. ABS language semantics

termination of the execution as well as the corresponding returned value. And finally, function $selectObj(S)$ selects an idle object from an idle cobox from a state S. Function $selectTask(q)$ selects a task from a task pool q. Both functions are left unspecified, such that different definitions correspond to different scheduling policies for handling tasks (see [BBS13]).

Intuitively, the rules in the semantics are as follows. Rules new-obj and new-cobox describe object and cobox creation, respectively. When the instruction being executed in the active task of an object o is a new or a newcog instruction, a new ob element is added to the program state with a reference to a fresh object o', a set of initialized fields a', no active task selected, and an empty task queue. Both rules differ in the cobox that the newly created object belongs to: in new-obj it is the same cobox to which o belongs, while in new-cobox it is a fresh cobox c'. Rule async-call handles asynchronous invocations to methods. In this case, a fresh future variable f is created with initial value $\bot$, and a new task with the instructions of the method invoked is added to the task queue of the object o' on which the method is invoked. Observe that o and o' may refer to the same object if o' is a reference to this (for brevity, we have not included an explicit rule to handle this case). The invoked method terminates its execution when a return instruction is reached. This is handled by rule return. The special local variable ret contains the identifier of the future variable that must be updated upon method return. The instruction await is handled by means of rules await-t and await-f. Rule await-t handles the case in which the future variable has a value already, updating the local variable accordingly, and continuing the execution. In contrast, rule await-f suspends the task executing the await instruction and adds it to the queue of pending tasks of the object. Observe that $idle$ becomes the active task of the object in order to allow that another task can be executed in the cobox. When the active object in a cobox has the $idle$ task, the cobox itself is released so that rule activate can select a task from the lists of pending tasks of the objects executing in that cobox.

Execution steps are denoted $S \rightsquigarrow^{b}_{o} S'$, indicating that we move from state S to state S' by executing instruction b on the object o. Traces take the form $t \equiv S_0 \rightsquigarrow^{b_0}_{o_\epsilon} \cdots \rightsquigarrow^{b_{n-1}}_{o_{n-1}} S_n$ where S_0 is an initial state of the form $\{ob(o_\epsilon, \bot, idle, \{\langle \bot, (f_\epsilon = this!\textsf{main}(\bar{v}))\rangle\}), cobox(c_\epsilon, idle), fut(f_\epsilon, \bot)\}$. Given a trace t, we use $steps(t)$ to denote the set of steps that form trace t. Since execution is non-deterministic, given a program $P(\bar{x})$, multiple (possibly infinite) fully expanded traces may exist. We use $executions(P(\bar{x}))$ to denote the set of all possible fully expanded traces for $P(\bar{x})$.

Example 2.1. Our running example in Figure 2 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. Method start initializes a data access object (DAO) that is used by Handler objects to request data from the database. Then, it creates n Handler objects at program point ② and starts their execution via the run method. The DAO object creates a fresh DB object at program point ④, that will actually execute queries from handlers. When executing run, each handler performs m

```
void main (Int n, Int m) {
① Server s = newcog Server(null);
    s!start (n,m);
    return;
}
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;
    }
    return;
  }
  void initDAO () {
③   this.dao = new DAO(null);
    Fut f<void> = this.dao!initDB();
    await f?;
    return;
  }
}
```

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

Fig. 2. Running Example.

requests to the DAO object by invoking method query. The use of Fut<void> variables and await instructions allows method synchronization as explained above. Regarding distribution, observe that, in addition to the initial cobox, the configuration contains a single distributed component (the Server cobox at ①), as all other objects are created using new.

A trace with some execution steps of the example program is shown in Figure 3. Each rule is labelled with the name of the rule of the semantics applied. The execution starts with an initial cobox c_ϵ and object o_ϵ on which method main is executed. When a newcog instruction is executed, a fresh cobox is created as well as a fresh object with *idle* as the active task, as shown in the program state of Step 1. An asynchronous call creates a new task in the pending tasks pool of the callee object: Step 2 shows a new task added to the queue of a different object, which is non-deterministically selected for execution in Step 4 by means of function $selectObj(S)$. In contrast, in Step 5 the newly created task is added to the pending tasks queue of the same object o_1 that performs the call. In both cases, the task is not activated until rule activate selects it from the pool. Observe that objects o_ϵ and o_1 can execute non-deterministically after Step 2, and this will not affect the results since fields are local to their objects (i.e., they can only be accessed through method calls). In other words, if Steps 3 and 4 are swapped in the trace, we have exactly the same execution but two different traces. On the other hand, different scheduling policies handled by *selectObj* and *selectTask* functions may yield to different execution traces with possibly different results. A cobox is released in two cases: when a return rule is applied (Step 3) or when an await instruction is reached and the future variable being awaited for has no value yet (await-f in Step 6). ■

3. Background: Points-to and Resource Analysis

In this article, we make use of the techniques of points-to analysis [MRR05, SBL11, APC12] and resource analysis [AAG11a, APC12] to infer quantified abstract configurations. A points-to and resource analyses for a language like ours, but without the instruction newcog, are defined and proved correct in [APC12].

$$
\begin{array}{lll}
 & & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \langle[\mathsf{s}\mapsto\bot, \mathit{ret}\mapsto f_\epsilon], (\mathsf{Server\ s} = \textbf{newcog}\ \mathsf{Server}(\textbf{null});\ldots)\rangle, \emptyset)\ \mathit{cobox}(c_\epsilon, o_\epsilon)\ \mathit{fut}(f_\epsilon, \bot) \\[6pt]
(1) & \leadsto_{o_\epsilon}^{\textsf{(new-cobox)}} & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \langle[\mathsf{s}\mapsto o_1, \mathit{ret}\mapsto f_\epsilon], (\mathsf{s!start(n,m)};\ldots)\rangle, \emptyset)\ \mathit{ob}(o_1, [\mathsf{dao}\mapsto \mathit{null}, \mathsf{cobox}\mapsto c_1], \mathit{idle}, \emptyset) \\
 & & \mathit{cobox}(c_\epsilon, o_\epsilon)\ \mathit{cobox}(c_1, \mathit{idle})\ \mathit{fut}(f_\epsilon, \bot) \\[6pt]
(2) & \leadsto_{o_\epsilon}^{\textsf{(async-call)}} & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \langle[\mathsf{s}\mapsto o_1, \mathit{ret}\mapsto f_\epsilon], \textbf{return;}\rangle, \emptyset) \\
 & & \mathit{ob}(o_1, [\mathsf{dao}\mapsto \mathit{null}, \mathit{cobox}\mapsto c_1], \mathit{idle}, \{\langle[\mathsf{n}\mapsto 2, \mathsf{m}\mapsto 2, \mathit{this}\mapsto o_1, \mathsf{f}\mapsto\bot, \mathit{ret}\mapsto f_1], (\mathsf{Fut\ f}<\textbf{void}>=\textbf{this}\mathsf{!initDAO()};\ldots)\rangle\}) \\
 & & \mathit{cobox}(c_\epsilon, o_\epsilon)\ \mathit{cobox}(c_1, \mathit{idle})\ \mathit{fut}(f_\epsilon, \bot)\ \mathit{fut}(f_1, \bot) \\[6pt]
(3) & \leadsto_{o_\epsilon}^{\textsf{(return)}} & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \mathit{idle}, \emptyset) \\
 & & \mathit{ob}(o_1, [\mathsf{dao}\mapsto \mathit{null}, \mathit{cobox}\mapsto c_1], \mathit{idle}, \{\langle[\mathsf{n}\mapsto 2, \mathsf{m}\mapsto 2, \mathit{this}\mapsto o_1, \mathsf{f}\mapsto\bot, \mathit{ret}\mapsto f_1], (\mathsf{Fut\ f}<\textbf{void}>=\textbf{this}\mathsf{!initDAO()};\ldots)\rangle\}) \\
 & & \mathit{cobox}(c_\epsilon, \mathit{idle})\ \mathit{cobox}(c_1, \mathit{idle})\ \mathit{fut}(f_\epsilon, \mathit{null})\ \mathit{fut}(f_1, \bot) \\[6pt]
(4) & \leadsto_{o_1}^{\textsf{(activate)}} & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \mathit{idle}, \emptyset) \\
 & & \mathit{ob}(o_1, [\mathsf{dao}\mapsto \mathit{null}, \mathit{cobox}\mapsto c_1], \langle[\mathsf{n}\mapsto 2, \mathsf{m}\mapsto 2, \mathit{this}\mapsto o_1, \mathsf{f}\mapsto\bot, \mathit{ret}\mapsto f_1], (\mathsf{Fut\ f}<\textbf{void}> = \textbf{this}\mathsf{!initDAO()};\ldots)\rangle, \emptyset) \\
 & & \mathit{cobox}(c_\epsilon, \mathit{idle})\ \mathit{cobox}(c_1, o_1)\ \mathit{fut}(f_\epsilon, \mathit{null})\ \mathit{fut}(f_1, \bot) \\[6pt]
(5) & \leadsto_{o_1}^{\textsf{(async-call)}} & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \mathit{idle}, \emptyset) \\
 & & \mathit{ob}(o_1, [\mathsf{dao}\mapsto \mathit{null}, \mathit{cobox}\mapsto c_1], \langle[\mathsf{n}\mapsto 2, \mathsf{m}\mapsto 2, \mathit{this}\mapsto o_1, \mathsf{f}\mapsto f_2, \mathit{ret}\mapsto f_1], (\textbf{await}\ \mathsf{f?};\ldots)\rangle, \\
 & & \quad \{\langle[\mathit{this}\mapsto o_1, \mathsf{f}\mapsto\bot, \mathit{ret}\mapsto f_2], (\textbf{this}.\mathsf{dao} = \textbf{new}\ \mathsf{DAO}(\textbf{null}))\rangle\}) \\
 & & \mathit{cobox}(c_\epsilon, \mathit{idle})\ \mathit{cobox}(c_1, o_1)\ \mathit{fut}(f_\epsilon, \mathit{null})\ \mathit{fut}(f_1, \bot)\ \mathit{fut}(f_2, \bot) \\[6pt]
(6) & \leadsto_{o_1}^{\textsf{(await-f)}} & \mathit{ob}(o_\epsilon, [\mathit{cobox}\mapsto c_\epsilon], \mathit{idle}, \emptyset) \\
 & & \mathit{ob}(o_1, [\mathsf{dao}\mapsto \mathit{null}, \mathit{cobox}\mapsto c_1], \mathit{idle}, \\
 & & \quad \{\langle[\mathit{this}\mapsto o_1, \mathsf{f}\mapsto\bot, \mathit{ret}\mapsto f_2], (\textbf{this}.\mathsf{dao} = \textbf{new}\ \mathsf{DAO}(\textbf{null}))\rangle, \langle[\mathsf{n}\mapsto 2, \mathsf{m}\mapsto 2, \mathit{this}\mapsto o_1, \mathsf{f}\mapsto f_2, \mathit{ret}\mapsto f_1], \\
 & & \quad (\textbf{await}\ \mathsf{f?};\ldots)\rangle\}) \\
 & & \mathit{cobox}(c_\epsilon, \mathit{idle})\ \mathit{cobox}(c_1, \mathit{idle})\ \mathit{fut}(f_\epsilon, \mathit{null})\ \mathit{fut}(f_1, \bot)\ \mathit{fut}(f_2, \bot)
\end{array}
$$

Fig. 3. Trace with some execution steps for the running example (with n=2 and m=2).

Handling newcog does not pose any further difficulty as it can be handled as new by both the points-to and resource analysis. In a subsequent step, we will give a special treatment to newcog in order to define the notion of quantified configuration. In what follows, we use points-to and resource analysis as black boxes as much as possible. Still, we need to describe the basic components that have to be used and/or adapted for our purposes.

3.1. Points-to Analysis

An essential concept of the resource analysis framework for distributed systems in [AAG11a, APC12] is the notion of *cost center*. A cost center is associated to every 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 [APC12]. The aim of points-to analysis is to approximate the set of objects (or coboxes) that each reference variable may point to during program execution. Let us introduce some notation. All instructions are labeled, such that $b \equiv q : i$ denotes that instruction b has q as label (program point) and i is the proper instruction. An *allocation site* is a program point where i is a new or newcog instruction. Following [MRR05, SBL11], we use the notion of *allocation sequence* to abstract the sequence of object creations. An allocation sequence is a syntactic construction of the form $o_{ab\ldots cd}$, where all elements in $ab\ldots cd$ are allocation sites, and it represents all run-time objects that were created at program point d when the enclosing instance method was invoked on an object represented by $o_{ab\ldots c}$, which in turn was created at allocation site c, and so on. Note that allocation sequences are not object identifiers since objects created within loops have the same allocation sequence as it will be seen in the following example.

Example 3.1 (allocation sequence). Figure 4 shows (part of) the result of applying points-to analysis to each program point. Object creations use the allocation sequences pointed to by this to generate new allocation sequences by adding the current allocation site. The analysis starts from the main method with *this* pointing to o_ϵ. At allocation site ① a new server is created, thus variable s points to object o_1. Since o_ϵ is not relevant for the analysis of the running example, in what follows, we omit it. The set of possible values

```
void main (Int n, Int m) {                 $\{this \mapsto \{o_\epsilon\}\}$
① Server s = newcog Server(null);          $\{this \mapsto \{o_\epsilon\}, s \mapsto \{o_1\}\}$
  s!start(n,m);                            $\{this \mapsto \{o_\epsilon\}, s \mapsto \{o_1\}\}$
  return;                                  $\{this \mapsto \{o_\epsilon\}, s \mapsto \{o_1\}\}$
}

 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;                             $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$
  }                                        $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$
  return;                                  $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}, h \mapsto \{o_{12}\}\}$
 }

 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?;                                $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
  return;                                  $\{this \mapsto \{o_1\}, o_1.dao \mapsto \{o_{13}\}\}$
 }

void initDB () {                           $\{this \mapsto \{o_{13}\}\}$
④ this.db = new DB();                      $\{this \mapsto \{o_{13}\}, o_{13}.db \mapsto \{o_{34}\}\}$
 }
```

Fig. 4. Fragment of points-to analysis with $k = 2$ applied to the running example

for this within a method comes from the object name(s) for the variable used to call the method. Then, at program point ② $this \mapsto \{o_1\}$, then the object created at ② is identified by the allocation sequence 12, that is, o_{12}. Observe that o_{12} represents multiple objects as they are created within a loop. ■

We will use o_l to refer to an object with allocation sequence l. As we will see in Section 4, we use multisets to handle different objects with the same allocation sequence. Allocation sequences have in principle unbounded length (e.g. recursion) and thus it is sometimes necessary to lose precision during analysis. Then, like in [MRR05], we limit the length of the allocation sequences to a constant value $k \geq 1$. Thus, k defines the maximum size of allocation sequences, and it allows controlling the precision of the analysis. The larger values of k, the more precise results are obtained by the points-to analysis. Let AS be the set of all allocation sites in a program. Given a value of $k \geq 1$, the analysis considers a finite set of *object names*, which is defined as $\mathcal{AS} = \{o_l \mid l \in \{\epsilon\} \cup AS \cup AS^2 \ldots AS^k\}$. 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. The size of the allocation sequence is therefore limited during the execution of the points-to analysis as follows. For an object name $o_{ab\ldots c}$ of length at most k and an allocation site d, we define the operation $ab\ldots c \oplus_k d$ as $ab\ldots cd$ if $|ab\ldots cd| \leq k$, or $b\ldots cd$ 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 with precision k by means of a function $pt(q, x)$, which returns the set of object names at program point q computed by the analysis for a given reference variable x. For the sake of readability, we omit k from the parameters of pt as it is fixed beforehand and remains constant for the whole analysis. In addition, we use $\mathcal{O}$ to refer to the set of all object names generated by the points-to analysis.

Example 3.2 (points-to analysis). For the running example, k=3 is the smallest k for which no information is lost when handling object names. If we apply the points-to analysis with $k = 3$, at program point ④ we have that $this \mapsto \{o_{13}\}$, thus we apply $13 \oplus_3 4$, which returns the sequence 134, and consequently $o_{13}.db \mapsto \{o_{134}\}$. However, if points-to analysis is applied with $k = 2$, at program point ④ the application of

```
void main2 (Int n, Int m) {
① Server s = newcog Server(null);
    s!start (n,m);
⑤ Server s2 = newcog Server(null);
    s2!start (n,m);
    return;
}
```

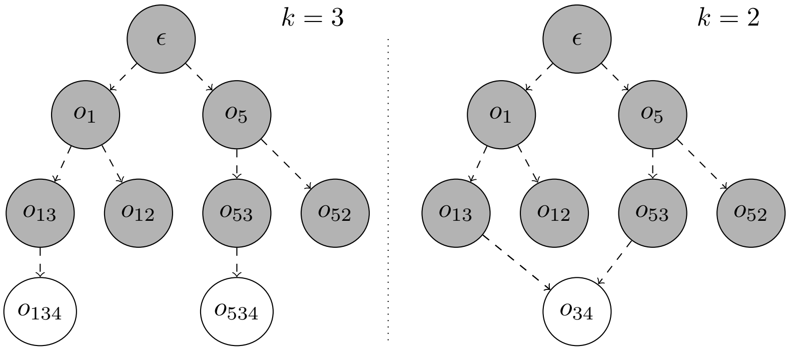


Fig. 5. Abstract configurations example with different values of k

$13 \oplus_2 4$ returns the sequence 34 because $|134| > 2$, and then $o_{13}.db \mapsto \{o_{34}\}$. As a consequence, the object name o_{34} obtained with $k = 2$ is losing information. Let us see another situation of precision loss. To the left of Figure 5, we define a main2 method that generates two Server objects, created at program points ① and ⑤. The graph in the middle and right will be explained later. At ④, we could have $this \mapsto \{o_{13}\}$ and $this \mapsto \{o_{53}\}$, thus at ④ we create two object names, o_{134} and o_{534}. If $k = 2$ both object names are abstracted to o_{34}.
For the next examples that use the running example, we use $k = 2$ and we have $\mathcal{O}=\{o_\epsilon, o_1, o_{12}, o_{13}, o_{34}\}$. ■

In what follows, in order to relate the concrete identifier for the objects used in the semantics to the abstract names inferred by the points-to analysis, we use $name(o_l)$ to refer to the abstract object name in $\mathcal{O}$ that represents the concrete object o and whose allocation sequence is l. Concretely, $name(o_l)$ is o_λ, where λ is the longest suffix of length at most k of the (possibly unbounded) allocation sequence l. As before, we do not include k as parameter of $name$.

Example 3.3 (name). For representing the concrete objects we use their allocation sequences. Thus, in the running example, the object created at ④ is represented by o_{134}. Function $name(o_{134})$ returns the object name that corresponds to o_{134}, that is o_{34} since $k = 2$. The function $name$ applied to o_{13} does not remove any allocation site, that is, $name(o_{13}) = o_{13}$. ■

3.2. Cost Models

Cost models determine the type of resource we are measuring. Traditionally, a cost model $\mathcal{M}$ is a function $\mathcal{M} : Instr \mapsto \mathbb{N}$ that for each instruction in the set of instructions $Instr$ returns a natural number that 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 $\mathcal{M}^{inst}$ that counts one unit for any instruction, i.e., $\mathcal{M}^{inst}(b) = 1$, for any $b \in Instr$. We use superscripts in $\mathcal{M}$ just as part of the name of a particular cost model for distinguishing the different cost models described throughout the paper.

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 the cost to the corresponding cost center. In particular, we use the results of the points-to analysis to determine the set of object names that might execute the corresponding instruction. A cost model that can be used for this purpose is defined as follows.

Definition 3.4 ($\mathcal{M}^I$ cost model). Given an instruction $b \equiv q : i$ in program P, and the results of the points-to analysis, the cost model $\mathcal{M}^I(b)$ is a function that returns $c(o_\lambda) * 1$, where $o_\lambda \in pt(q, this)$.

As before, we count "1" instruction but now we attribute it to the cost center, that corresponds to one object that executes the instruction, i.e. $this$. In order to obtain an upper bound on the cost, all possible object names returned by $pt(q, this)$ are covered by generating cost equations for each $c(o_\lambda)$ (for more details, see [APC12]). In the previous definition, $c(\_)$ is a symbolic artifact that we include in the cost expressions so as to attribute the cost to the corresponding cost center. Then, to obtain how many instructions have been executed by the cost center $c(o_\lambda)$, we replace $c(o_\lambda)$ by 1 and $c(o')$ by 0 for all other $o' \neq o_\lambda$.

Given a trace step $S_i \leadsto^b_{o_l} S_{i+1}$, we will use $\mathcal{M}(S_i \leadsto^b_{o_l} S_{i+1})$ to refer to $\mathcal{M}(b)$, i.e., the cost of executing instruction b with respect to a cost model $\mathcal{M}$. The actual total cost of a trace t within object o_l, is defined as $cost_P(t, o_l, \mathcal{M}) = \sum_{s \in steps(t)} \mathcal{M}(s)|_{\{name(o_l)\}}$, where we use $\mathcal{M}(s)|_{\mathcal{N}}$ to denote the result of replacing each cost center $c(o_\lambda)$ by 1 if $o_\lambda \in \mathcal{N}$ and by 0 otherwise. As we have seen, for the cost model $\mathcal{M}^I$, the symbolic cost expression $c(o_\lambda)$ includes a single object name, however, the artifact $c(\_)$ can include more than one argument, as will see later. In the following sections, we will define the cost models and the cost centers that we use for our analysis.

3.3. Upper Bounds

Given a program $P(\overline{x})$, where $\overline{x}$ are the input values for the arguments of the main method, and a definition of cost model $\mathcal{M}$, the resource analysis obtains an *upper bound* $UB_P^{\mathcal{M}}(\overline{x})$ that is an expression of the form $c_1{\cdot}e_1 + \ldots + c_n{\cdot}e_n$ where c_i are cost centers and e_i are cost expressions (e.g., polynomials, exponential functions, etc.) with $i = 1, \ldots, n$. For the cost model $\mathcal{M}^{\mathcal{I}}$, the upper-bound expression $UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})$ is an expression of the form $c(o_{\lambda_1}){\cdot}e_1 + \ldots + c(o_{\lambda_n}){\cdot}e_n$ where each $c(o_{\lambda_i}){\cdot}e_i$ represents that the object o_{λ_i} executes the number of instructions e_i. By replacing $c(o_{\lambda_i})$ by 1 and all other cost centers by 0 we obtain an upper bound on the number of instructions performed in the object o_{λ_i}. We use $UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})|_{\mathcal{N}}$ to denote the result of replacing $c(o_\lambda)$ by 1 if $o_\lambda \in \mathcal{N}$ and by 0 otherwise.

Example 3.5. 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

$$UB_{main}^{\mathcal{M}^I}(n,m) = c(o_1){\cdot}18{+}c(o_{13}){\cdot}6{+}c(o_1){\cdot}12{\cdot}\mathsf{nat}(n){+}$$

$$c(o_{12}){\cdot}6{\cdot}\mathsf{nat}(n){+}c(o_{12}){\cdot}8{\cdot}\mathsf{nat}(n){\cdot}\mathsf{nat}(m){+}c(o_{13}){\cdot}2{\cdot}\mathsf{nat}(n){\cdot}\mathsf{nat}(m){+}c(o_{34}){\cdot}\mathsf{nat}(n){\cdot}\mathsf{nat}(m),$$

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:

$$UB_{main}^{\mathcal{M}^I}(n,m)|_{\{o_{12}\}} = n \cdot (6 + 8 \cdot m)$$

This upper bound states that the objects created at program point ② from the Server object created at program point ① execute at most that number of instructions. Observe that object name o_{12} refers to several concrete objects. Intuitively, n in the upper bound expression corresponds to the number of objects created at program point ②, which is the maximum number of iterations that the loop in method start performs. The expression enclosed in parenthesis corresponds to the maximum number of instructions executed by each Handler object in method run. This expression is linear with respect to m, since the loop in method run iterates m times. The constants 6 and 8 refer to the concrete number of instructions executed. We note that these numbers do not correspond directly to the source level instructions since they are put in some normal form before the analysis starts (e.g., an arithmetic expression involving several operations uses auxiliary variables to perform each individual operation and such auxiliary variable is used in the next operation, etc.). The analyzer counts the normalized instructions instead of the source level ones. ■

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 by the cost center that c_i represents. Formally, the following theorem from [APC12] states the soundness result of the resource analysis.

Theorem 3.6 (soundness [APC12]). Let P be a program with input values $\overline{x}$ and S_0 its initial state. If $t \equiv S_0 \leadsto \ldots \leadsto S_n$ is an execution trace, then for any object o_l such that $ob(o_l, \_, \_, \_) \in S_i$, $0 \leq i \leq n$, it holds that

$$cost_P(t, o_l, \mathcal{M}) {\leq} UB_{\mathsf{main}}^{\mathcal{M}}(\overline{x})|_{\{name(o_l)\}}$$

It should be noted that we omit explanations on the use of *norms* in the resource analysis (see, e.g., [AAG07, BCG07]) because they are orthogonal to our contribution. Norms are used to determine the notion of size that the resource analysis relies on. For instance, when a loop traverses a list, its resource consumption typically depends on the length of such list. Thus, the resource analysis will give the upper bound in terms of the size of the list, in such a way that the variables used in the upper bounds represent the sizes of the corresponding data (abstracted using some norm). In our examples, the resource consumption only depends on integer data. In such cases, the norm used by the analysis coincides with the value of the integer variable. Hence, there is no ambiguity and we can ignore the use of norms in what follows as it does not affect our approach.

Example 3.7. Although the resource analysis in [AAG11a, APC12] 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:

$$UB^{\mathcal{M}^I}_{main}(n,m)|_{\{o_1,o_{12}\}} = 18 + n \cdot (18 + 8 \cdot m)$$

This expression is an upper bound of the total number of instructions executed by the objects created at ① and the objects created from these objects at ②. If objects with names o_1 and o_{12} are located in the same cobox, this is an upper bound on the number of instructions executed in that cobox. The first constant with value 18 in the upper bound expression corresponds to the (normalized) instructions executed by object name o_1 in method start that are outside the while loop, while the second constant with value 18 refers to the instructions executed inside that loop: 12 of them by object name o_1, and 6 of them by object name o_{12}, as shown above in the upper bound computed for o_{12}. ■

4. Concrete Definitions in Distributed Systems

As our first contribution, 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. As we have already mentioned, 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 object o_l, we use $root(o_l)$ to refer to the root object of the cobox that owns o_l. It can be defined as the object whose allocation sequence is the longest prefix of l that ends in an allocation site for coboxes, i.e., one site at which a newcog instruction is executed. Therefore, if l ends in an allocation site for coboxes, then $root(o_l) = o_l$. If it ends in an allocation site for objects, i.e., one where a new instruction is executed, then $root(o_{ab\ldots cd}) = root(o_{ab\ldots c})$. Given a trace t, the multiset of cobox roots created during t is defined as $cobox\_roots(t) = \{\!\!|o_l \mid \_ \leadsto^{q:\mathsf{newcog}\_}_{o_{j\ldots p}} \_ \in steps(t) \wedge\ l = j\ldots pq|\!\!\}$. Also, given a cobox root o_l, the multiset of objects it owns in a trace t is defined as $obj\_in\_cobox(o_l,t) = \{\!\!|o_{j\ldots pq}|\_ \leadsto^{q:\mathsf{new}}_{o_{j\ldots p}}\ \text{-}\ \_ \in steps(t) \wedge root(o_{j\ldots p}) = o_l|\!\!\}$.

Example 4.1. Deliberately, the running example shown in Figure 2 executes in a single cobox, in addition to the initial cobox c_ϵ (see Example 2.1) that executes main (i.e., all objects conceptually share the processor). It can be configured as a distributed application by creating coboxes instead of objects, i.e., by replacing selected new instructions by newcog and thus assuming that we have a new processor to execute the corresponding code. The graphs in Figure 6 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, e.g., for setting 1 we have two cobox roots, o_1 and o_{134}. Dashed edges represent the creation sequence, that is, object o_1 creates objects o_{13} and o_{12}, while object o_{13} creates object o_{134}. 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. As we will define in Definition 4.2, the configurations for the different settings capture textually the information in the graphs:

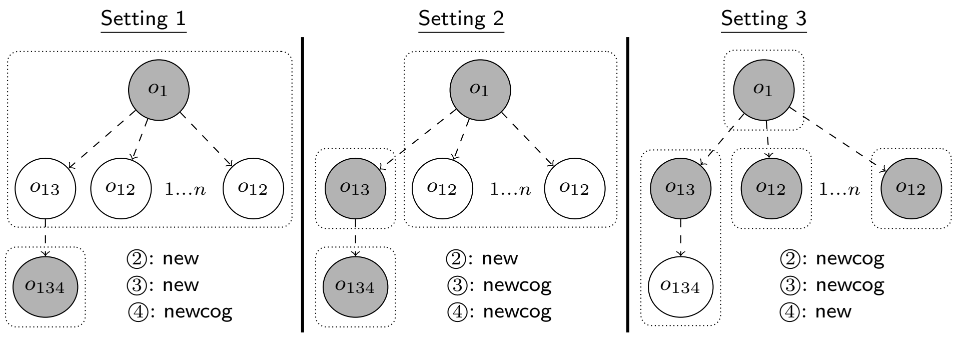


Fig. 6. Configurations for the running example for all settings.

$$\text{Setting 1:}\{\!|\langle o_1, \{\!|\underbrace{o_{12}, ..., o_{12}}_{n \text{ objects}}, o_{13}|\!\}\rangle, \langle o_{134}, \{\!||\!\}\rangle|\!\}$$

$$\text{Setting 2:}\{\!|\langle o_1, \{\!|\underbrace{o_{12}, ..., o_{12}}_{n \text{ objects}}|\!\}\rangle, \langle o_{13}, \{\!||\!\}\rangle, \langle o_{134}, \{\!||\!\}\rangle|\!\}$$

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

■

Definition 4.2 (configuration). Given an execution trace t, we define its *configuration*, denoted C_t, as:

$$C_t = \{\!|\langle o_l, obj\_in\_cobox(o_l, t)\rangle \mid o_l \in cobox\_roots(t)|\!\}$$

The *configuration* of a program P on input values $\overline{x}$, denoted $Conf_P(\overline{x})$ is defined as:

$$Conf_P(\overline{x}) = \{C_t \mid t \in executions(P(\overline{x}))\}.$$

For given input values and an allocation sequence, now we want to count the maximum number of instances (objects) created at such allocation sequence.

Example 4.3. 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. ■

We use $card(x, M)$ to refer to the cardinality of x in the multiset M.

Definition 4.4 (number of instances). Given an object o_l, a program P and input values $\overline{x}$, we define the *number of instances* for o_l as:

$$inst(o_l, P, \overline{x}) = \max_{t \in executions(P(\overline{x}))}(card(o_l, \{\!|o_{l'} \mid \_ \overset{q:i}{\rightsquigarrow}_{o_{j...p}} \_ \in steps(t) \wedge (i{\equiv}\mathsf{new}\ \_ \vee i{\equiv}\mathsf{newcog}\ \_) \wedge l'{=}j\ldots pq|\!\}))$$

4.2. Communication

The *communication* refers to the *interactions* between objects occurring during the execution of a program. As in the above section, objects are represented using allocation sequences. 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 that occurs in the trace t, and whose arcs, annotated with the method name are given by the interactions among objects.

Example 4.5. The interactions for any execution of our running example, and thus, the communication of the program, is depicted graphically in Figure 7, and, it is textually defined as:

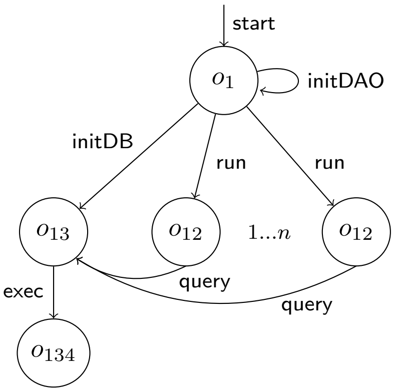


Fig. 7. Interactions for the running example

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

$$\underbrace{\langle o_{12}, o_{13}, \mathsf{query}\rangle, ..., \langle o_{12}, o_{13}, \mathsf{query}\rangle}_{n \cdot m \text{ interactions}}, \underbrace{\langle o_{13}, o_{134}, \mathsf{exec}\rangle, ..., \langle o_{13}, o_{134}, \mathsf{exec}\rangle}_{n \cdot m \text{ interactions}} |\!\}$$

■

Definition 4.6 (communication). Given an execution trace t, its *interactions*, denoted I_t, are defined as:

$$I_t = \{\!|\langle o_l, o_{l'}, m\rangle \mid S \leadsto_{o_l}^{q:x!m(\_)} \_ \in steps(t) \wedge ob(o_l, \_, \langle tv, \_\rangle, \_) \in S \wedge (x \mapsto o_{l'}) \in tv|\!\},$$

and the *communication* performed in the execution of a program P and input values $\overline{x}$, denoted $Comm_P(\overline{x})$ is defined as:

$$Comm_P(\overline{x}) = \{I_t \mid t \in executions(P(\overline{x}))\}.$$

Observe that the communication of the program includes 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.

Example 4.7. 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}. ■

Definition 4.8 (number of interactions). Given a program P and its input values $\overline{x}$, two objects o_l, $o_{l'}$, and a method m, we define the *number of interactions* from o_l to $o_{l'}$ by method m in the execution of P on $\overline{x}$ as: $ninter(o_l, o_{l'}, m, P, \overline{x}) = \max\limits_{I_t \in Comm_P(\overline{x})} (card(\langle o_l, o_{l'}, m\rangle, I_t))$.

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 [AAG11a, APC12] 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 that 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.

Example 5.1. The graph in Fig. 8 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_{34}. ■

Definition 5.2 (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 $V = \mathcal{O}$ and set of edges is $E = \{o_\lambda \rightarrow o_{\lambda'} \mid q{:}y{=}\mathsf{new}\ \_$ or $q{:}y{=}\mathsf{newcog}\ \_ \in P \wedge o_\lambda \in pt(q, \mathit{this}) \wedge o_{\lambda'} \in pt(q, y)\}$.

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 abstract cobox roots and find the set of object names that belong to the coboxes associated to such roots. Note that given an object name $o_{ab\ldots q}$ it can be decided whether it represents a cobox root, denoted $\mathit{is\_root}(o_{ab\ldots q})$, by simply checking whether the allocation site q contains a newcog instruction.

Example 5.3 (abstract configuration). The abstract configuration for the concrete setting 2 is represented graphically in Figure 8. As before, cobox roots appear in grey and objects are grouped by cobox (dotted rectangles). The abstract configurations for the settings in Example 4.1 are:

$$\begin{array}{ll}\text{Setting 1:} & \langle o_1, \{o_{12}, o_{13}\}\rangle,\ \langle o_{34}, \{\}\rangle,\\ \text{Setting 2:} & \langle o_1, \{o_{12}\}\rangle,\ \langle o_{13}, \{\}\rangle,\ \langle o_{34}, \{\}\rangle,\\ \text{Setting 3:} & \langle o_1, \{\}\rangle,\ \langle o_{12}, \{\}\rangle,\ \langle o_{13}, \{o_{34}\}\rangle\end{array}$$

■

We write $x \rightsquigarrow y$ to indicate that there is a non-empty path in a graph from x to y and we denote by $\mathit{interm}(x, y)$ the set of intermediate nodes in the path (excluding x and y).

Definition 5.4 (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 $\langle o_\lambda, \mathit{obj\_names\_in\_cobox}(o_\lambda, G_P)\rangle$ s.t. $o_\lambda \in V \wedge \mathit{is\_root}(o_\lambda)$ where $\mathit{obj\_names\_in\_cobox}(o_\lambda, G_P) = \{o_{\lambda'} \in V$ s.t. $o_\lambda \rightsquigarrow o_{\lambda'}$ in G_P and $\forall o_{\lambda''} \in \mathit{interm}(o_\lambda, o_{\lambda'})\ \wedge\ \neg\mathit{is\_root}(o_{\lambda''})\}$.

Note that, in the above definition, function $\mathit{obj\_names\_in\_cobox}(o_\lambda, G_P)$ returns the subset of object names that are part of the cobox whose root is the object name o_λ in the points-to graph G_P.

Example 5.5. Let us illustrate how the precision of the points-to analysis leads to different abstract configurations. To the right of Figure 5, we show the points-to graph obtained for method main2 (left of Figure 5) with different precision of the points-to analysis, $k = 2$ and $k = 3$ (using setting 3). It can be seen that the only difference in the graphs is that, for $k = 3$ we have nodes o_{134} and o_{534}, whereas for $k = 2$, due to the length bound in the object names, they are merged in only one node, o_{34}. Thus, we have different abstract configurations, for $k = 3$ we have coboxes $\langle o_{13}, \{o_{134}\}\rangle$ and $\langle o_{53}, \{o_{534}\}\rangle$, while for $k = 2$ we have $\langle o_{13}, \{o_{34}\}\rangle$ and $\langle o_{53}, \{o_{34}\}\rangle$. Note that for $k = 2$ node o_{34} belongs to two different coboxes, which is a less precise abstract configuration than the one for $k = 3$. ■

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 a set of objects O and a set of object names $\mathcal{N}$, we write $\mathit{covers}(\mathcal{N}, O)$ if $\forall o_l \in O, \exists\ o_\lambda \in \mathcal{N}$ s.t. $o_\lambda = \mathit{name}(o_l)$. Given $\langle o_l, O\rangle$ and $\langle o_\lambda, \mathcal{N}\rangle$, we write $\mathit{covers}(\langle o_\lambda, \mathcal{N}\rangle, \langle o_l, O\rangle)$ if $o_\lambda = \mathit{name}(o_l)$ and $\mathit{covers}(\mathcal{N}, O)$. Given a trace t, its concrete configuration C_t, and an abstract configuration $\mathcal{A}$, we write $\mathit{covers}(\mathcal{A}, C_t)$ if $\forall \langle o_l, O\rangle \in C_t$ there exists $\langle o_\lambda, \mathcal{N}\rangle \in \mathcal{A}$ s.t. $\mathit{covers}(\langle o_\lambda, \mathcal{N}\rangle, \langle o_l, O\rangle)$.

Theorem 5.6 (soundness of abstract configurations). Let P be a program and $\mathcal{A}_P$ its abstract configuration. Then $\forall \overline{x}, \forall C_t \in \mathit{Conf}_P(\overline{x})$, $\mathit{covers}(\mathcal{A}_P, C_t)$ holds.

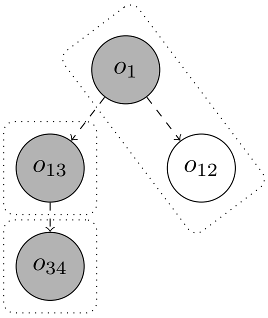


Fig. 8. Points-to graph and abstract configuration for the running example with setting 2

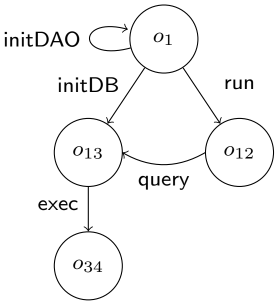


Fig. 9. Interactions graph for the running example

Proof. (sketch) The proof relies on the fact that the underlying points-to analysis of Section 3.1 is correct (a correctness proof for the points-to analysis can be found in [APC12]). Then, we let C_t be a multiset of pairs $\langle o_l, \mathit{obj\_in\_cobox}(o_l,t)\rangle$, where l is an allocation sequence for a cobox root (created with newcog). Let $\mathit{obj\_in\_cobox}(o_l,t)$ be the multiset of objects (transitively) created with new from o_l. Let $\mathcal{A}_p$ be a set of pairs $\langle o_\lambda, \mathit{obj\_names\_in\_cobox}(o_\lambda, G_p)\rangle$, where o_λ is an object name for a cobox root. Finally, let $\mathit{obj\_names\_in\_cobox}(o_\lambda, G_p)$ be the set of object names reachable from o_λ by a path in G_p in which there are no cobox roots except o_λ. Now, we proceed to prove that for every $\langle o_l, O\rangle \in C_t$, there exists $\langle o_\lambda, \mathcal{N}\rangle \in \mathcal{A}_p$ such that $\mathit{covers}(\langle o_\lambda, \mathcal{N}\rangle, \langle o_l, O\rangle)$ holds.

By construction of the points-to graph, we have that the set of vertices V in the graph contains all object names that have been generated during the points-to analysis $\mathcal{O}$. Since this analysis is correct, every possible allocation sequence in an execution trace is covered by some object name. In particular, for every $\langle o_l, s\rangle \in C_t$, there exists $\langle o_\lambda, s'\rangle \in \mathcal{A}_p$ such that $o_\lambda{=}\mathit{name}(o_l)$ in $\mathcal{O}$. Analogously, for any trace t and object o_l in t such that $o_l{=}\mathit{root}(o_l)$, we have that $\mathit{covers}(\mathit{obj\_names\_in\_cobox}(o_\lambda, G_p), \mathit{obj\_in\_cobox}(o_l,t))$ holds, since all allocation sequences in $\mathit{obj\_in\_cobox}(o_l,t)$ have their corresponding object names in $\mathcal{O}$. Let $o_{l'}$ be an element of $\mathit{obj\_in\_cobox}(o_l,t)$. We distinguish two cases:

- If $|l'| \leq k$, then l' is the result of appending some allocation sites to l, and therefore the object name $o_{\lambda'}{=}\mathit{name}(o_{l'})$ coincides with its allocation sequence: $l' = \lambda'$.
- Otherwise $|l'| > k$, and then λ' in $o_{\lambda'}{=}\mathit{name}(o_{l'})$ is the suffix of l' of length k.

It is straightforward to prove by induction on the allocation sites in l' that are not in l that there exists a path in G_p from o_λ to $o_{\lambda'}$ that has no intermediate vertex that is a cobox root, and therefore $o_{\lambda'} \in \mathit{obj\_names\_in\_cobox}(o_\lambda, G_p)$. □

It is easy to see that the theorem holds for the configuration Conf_P in Example 4.1, and any abstract configuration $\mathcal{A}_P$ of Example. 5.3.

(Non-quantified) abstract configurations are already useful when combined with the resource analysis in Section 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_λ, we use $\mathit{cobox}(o_\lambda, G_P)$ to denote $\{o_\lambda\} \cup \mathit{obj\_names\_in\_cobox}(o_\lambda, G_P)$.

Example 5.7. Using the UB expression inferred in Example 3.5 and the abstract configurations for all settings in Example 5.3, we can obtain the cost for each cobox name. The following table shows the results obtained from $\mathit{UB}^{\mathcal{M}^I}_{\mathit{main}}(n,m)|_{\mathit{cobox}(c,G_{\mathit{main}})}$ where c corresponds, in each case, to the cobox name in the considered abstract configuration:

Setting 1		Setting 2		Setting 3	
cobox	*UB*	*cobox*	*UB*	*cobox*	*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_{34}	$n{\cdot}m$	o_{13}	$6 + 2{\cdot}n{\cdot}m$	o_{12}	$6{\cdot}n + 8{\cdot}n{\cdot}m$
		o_{34}	$n{\cdot}m$	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 $\mathcal{M}^C$ cost model as follows.

Definition 5.8 ($\mathcal{M}^C$ cost model). Given an instruction b in program P and the points-to analysis results for P with precision k, the cost model $\mathcal{M}^C(b)$ is a function that returns $c(o_{\lambda'})$ if $b \equiv q{:}y{=}\mathsf{new}\ C$ or $b \equiv q{:}y{=}\mathsf{newcog}\ C$, and 0 otherwise, where $o_\lambda \in pt(q, \mathit{this})$ and $\lambda' = \lambda \oplus_k q$.

The novelty is on how the information computed by the points-to analysis is used in the cost model: it concatenates, by means of the operator $\oplus_k$, the name of the object that corresponds to the considered object name for this with the instruction allocation site q. The cost center $c(o_{\lambda'})$ allows counting the elements created at program point q for each particular instance of this considered by the points-to analysis.

Example 5.9. Using $\mathcal{M}^C$, the upper bound obtained by the resource analysis in Sec. 3 for the configuration of the running example is the expression:

$$UB^{\mathcal{M}^C}_{main}(n,m) = c(o_1) + c(o_{13}) + c(o_{34}) + n \cdot c(o_{12})$$

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 $UB^{\mathcal{M}^C}_{main}(n,m)|_{\{cobox\}}$ where *cobox* is the object name of the objects identified as coboxes for each setting. For instance, for setting 1 we have two coboxes, o_1 and o_{34}, and for setting 2, we have o_1, o_{13} and o_{34}. The following table shows the UBs obtained for all settings:

Setting 1		Setting 2		Setting 3	
cobox	*UB*	*cobox*	*UB*	*cobox*	*UB*
o_1	1	o_1	1	o_1	1
o_{34}	1	o_{13}	1	o_{12}	n
		o_{34}	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 5.10 (soundness). Let P be a program with input values $\overline{x}$ and S_0 its initial state. If $t \equiv S_0 \leadsto \ldots \leadsto S_n$ is an execution trace, then for any object o_l such that $ob(o_l, \_, \_, \_) \in S_i$, $0 \le i \le n$, it holds that

$$inst(o_l, P, \overline{x}) \le UB^{\mathcal{M}^C}_P(\overline{x})|_{\{name(o_l)\}}.$$

Proof. (sketch) Soundness is derived from the following facts:

(a) By applying Theorem 3.6 with respect to the cost model $\mathcal{M}^C$, we have that

$$\forall \overline{x}, \forall t \in executions(P(\overline{x})), cost_P(t, o_l, \mathcal{M}^C) \le UB^{\mathcal{M}^C}_P(\overline{x})|_{\{name(o_l)\}}.$$

Since this inequality holds for all traces, the following also holds:

$$\max_{t \in executions(P(\overline{x}))} (cost_P(t, o_l, \mathcal{M}^C)) \le UB^{\mathcal{M}^C}_P(\overline{x})|_{\{name(o_l)\}}. \quad (1)$$

(b) Definition 4.4 states that *inst* is defined as

$$inst(o_l, P, \overline{x}) = \max_{t \in executions(P(\overline{x}))} (card(o_l, I))$$

where

$$I = \{\!|o_{l'} \mid \_ \leadsto^{q:i}_{o_{j\ldots p}} \_ \in \mathit{steps}(t) \wedge (i{\equiv}\mathsf{new}\ \_ \vee i{\equiv}\mathsf{newcog}\ \_) \wedge l'{=}j\ldots pq|\!\}.$$

Given the multiset of objects I, we use $\mathit{name}(I)$ to refer to the following multiset: $\{\!|o_\lambda \mid o_l \in I \wedge o_\lambda = \mathit{name}(o_l)|\!\}$. The following statement holds:

$$\mathit{card}(o_l, I) \leq \mathit{card}(\mathit{name}(o_l), \mathit{name}(I)).$$

As $\mathit{cost}_P(t, o_l, \mathcal{M}^C) = \sum_{s \in \mathit{steps}(t)} \mathcal{M}^C(s)|_{\{\mathit{name}(o_l)\}}$ (see Section 3.2), where $\mathcal{M}^C(s)|_{\{\mathit{name}(o_l)\}}$ counts 1 if $\mathcal{M}^C(s)$ returns $c(\mathit{name}(o_l))$ and 0 otherwise, $\mathit{cost}_P(t, o_l, \mathcal{M}^C) = \mathit{card}(\mathit{name}(o_l), \mathit{name}(I))$ holds. Therefore,

$$\mathit{inst}(o_l, P, \overline{x}) \leq \max_{t \in \mathit{executions}(P(\overline{x}))} (\mathit{cost}_P(t, o_l, \mathcal{M}^C)). \tag{2}$$

From (1) and (2), Theorem 5.10 holds. □

5.2. Quantified Communication

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

Example 5.11. Figure 9 shows the interaction graph for the running 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. Besides, object o_1 executes method initDAO and it calls initDB in the object o_{13}. Note that the multiple calls to query from o_{12} to o_{13} are abstracted by one edge. ■

Definition 5.12 (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 $V = \mathcal{O}$ and a set of edges $E = \{o_\lambda \xrightarrow{m} o_{\lambda'} \mid q{:}x!m(\_) \wedge o_\lambda \in pt(q, \mathit{this}) \wedge o_{\lambda'} \in pt(q, x)\}$.

We now integrate quantitative information in the interaction graph. For this purpose, we define a new cost model.

Definition 5.13 ($\mathcal{M}^K$ cost model). Given an instruction b in program P and the points-to analysis results for P, the cost model $\mathcal{M}^K(b)$ is a function that returns $c(m){\cdot}c(o_\lambda, o_{\lambda'})$ if $b \equiv x!m(\_)$, and 0 otherwise, where $o_\lambda \in pt(q, \mathit{this})$ and $o_{\lambda'} \in pt(q, x)$.

The key point is that for capturing interactions between objects, when applying the cost model to an instruction, we pass as parameters the considered object names of the caller and callee objects. The resulting upper bounds will contain cost centers made up of pairs of object names $c(o_\lambda, o_{\lambda'})$, where o_λ is the object that is executing and $o_{\lambda'}$ 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). In order to obtain upper bounds using this cost model, all combinations of o_λ and $o_{\lambda'}$ are tried by the underlying resouce analyzer. Soundness of the analysis guarantees that the upper bound is the maximum of all possibilities.

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 $c(o_\lambda, o_{\lambda'})$ 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 $c(o_\lambda, o_{\lambda'})$ by 1. (5) Furthermore, we can obtain the interactions between the distributed nodes by replacing by 1 those cost centers in which o_λ and $o_{\lambda'}$ 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.

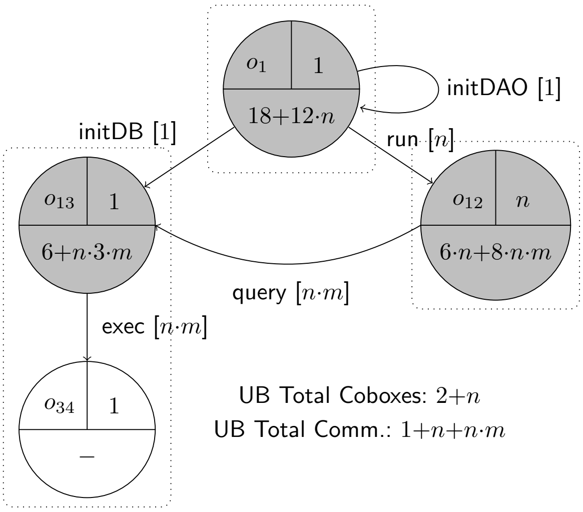


Fig. 10. QA for the running example for setting 3

Example 5.14. The interaction UB obtained by the resource analysis is as follows:

$$\begin{aligned} UB^{\mathcal{M}^K}_{main}(n,m) \quad = \quad & c(\mathsf{start})\cdot c(o_\epsilon,o_1)+c(\mathsf{initDAO})\cdot c(o_1,o_1)+c(\mathsf{initDB})\cdot c(o_1,o_{13})+ \\ & n\cdot c(\mathsf{run})\cdot c(o_1,o_{12})+n\cdot m\cdot c(\mathsf{query})\cdot c(o_{12},o_{13})+n\cdot m\cdot c(\mathsf{exec})\cdot c(o_{13},o_{34}) \end{aligned}$$

From this global UB, we obtain the following UBs on the number of interactions between coboxes for the different settings in Figure 6:

Setting 1			Setting 2			Setting 3		
method	*coboxes*	*UB*	*method*	*coboxes*	*UB*	*method*	*coboxes*	*UB*
exec	$o_1 \rightarrow o_{34}$	$n\cdot m$	query exec	$o_1 \rightarrow o_{13}$ $o_{13} \rightarrow o_{34}$	$n\cdot m$ $n\cdot m$	run initDB query	$o_1 \rightarrow o_{12}$ $o_1 \rightarrow o_{13}$ $o_{12} \rightarrow o_{13}$	n 1 $n\cdot m$
$n\cdot m$			$n\cdot m + n\cdot m$			$1+n+n\cdot m$		

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 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 Figure 10. Each node contains as object identifier its object name 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 Example 5.7). 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} interact with o_{13} $n\cdot m$ times calling method query). ■

From the example, we can figure out the applications described in Section 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 [AAG11a, APC12] (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.

We use $UB^{\mathcal{M}^K}_P(\overline{x})|_{N,M}$, where $N \subseteq \mathcal{O} \times \mathcal{O}$ is a set of pairs of object names and M is a set of method names, to denote the result of replacing in the resulting UB the expression $c(o_\lambda,o_{\lambda'})$ by 1 if $(o_\lambda,o_{\lambda'}) \in$

N and by 0 otherwise and the expression $c(m)$ by 1 if $m \in M$ and by 0 otherwise. Similarly, we use $cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K)$ to extend the definition given in Section 3.2 for the cost of a trace with respect to $(o_l, o_{l'})$ and m.

Theorem 5.15 (soundness). Let P be a program with input values $\overline{x}$ and S_0 its initial state. If $t \equiv S_0 \leadsto \ldots \leadsto S_n$ is an execution trace, then for any two objects o_l and $o_{l'}$ such that $ob(o_l, \_, \_, \_) \in S_i$, $ob(o_{l'}, \_, \_, \_) \in S_j$, $0 \leq i \leq n, 0 \leq j \leq n$, it holds that

$$ninter(o_l, o_{l'}, m, P, \overline{x}) \leq UB_P^{\mathcal{M}^K}(\overline{x})|_{\{(name(o_l), name(o_{l'})\}, \{m\}}.$$

Proof. (sketch) Soundness is derived from the following facts:

(a) By applying Theorem 3.6 to $\mathcal{M}^K$, we have that

$$\forall \overline{x}, \forall t \in executions(P(\overline{x})), cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K) \leq UB_P^{\mathcal{M}^K}(\overline{x})|_{\{(name(o_l), name(o_{l'}))\}, \{m\}}.$$

Since this inequality holds for all traces, the following also holds:

$$\max_{t \in executions(P(\overline{x}))} (cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K)) \leq UB_P^{\mathcal{M}^K}(\overline{x})|_{\{(name(o_l), name(o_{l'}))\}, \{m\}}. \qquad (3)$$

(b) Definition 4.8 states that *ninter* is defined as

$$ninter(o_l, o_{l'}, m, P, \overline{x}) = \max_{I_t \in Comm_P(\overline{x})} (card(\langle o_l, o_{l'}, m\rangle, I_t))$$

where $I_t = \{\!|\langle o_l, o_{l'}, m\rangle \mid S \leadsto_{o_l}^{q:x!m(\_)} \_ \in steps(t) \wedge ob(o_l, \_\,, \langle tv, \_\rangle, \_) \in S \wedge o_{l'} = tv(x)|\!\}$.

Since every trace $t \in executions(P(\overline{x}))$ is represented by an element $I_t \in Comm_P(\overline{x})$ (see Definition 4.6), the following holds:

$$ninter(o_l, o_{l'}, m, P, \overline{x}) = \max_{t \in executions(P(\overline{x}))} (card(\langle o_l, o_{l'}, m\rangle, I_t)).$$

Given I_t, we use $name(I_t)$ to refer to the following multiset:

$$\{\!|\langle o_\lambda, o_{\lambda'}, m\rangle \mid \langle o_l, o_{l'}, m\rangle \in I_t \wedge o_\lambda = name(o_l) \wedge o_{\lambda'} = name(o_{l'})|\!\}.$$

The following statement holds:

$$card(\langle o_l, o_{l'}, m\rangle, I_t) \leq card(\langle name(o_l), name(o_{l'}), m\rangle, name(I_t))$$

As $cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K) = \sum_{s \in steps(t)} \mathcal{M}^K(s)|_{\{(name(o_l), name(o_{l'}))\}, \{m\}}$ (see Section 3.2), where $\mathcal{M}^K(s)|_{\{(name(o_l), name(o_{l'}))\}, \{m\}}$ accounts 1 if $\mathcal{M}^K(s)$ returns $c(name(o_l), name(o_{l'})) * c(m)$ and 0 otherwise, $cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K) = card(name(\langle o_l, o_{l'}, m\rangle), name(I_I))$ holds. Therefore,

$$ninter(o_l, o_{l'}, m, P, \overline{x}) \leq \max_{t \in executions(P(\overline{x}))} (cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K)). \qquad (4)$$

From (3) and (4), Theorem 5.15 holds. □

6. Finding Optimal Settings for Distributed Systems

As we have seen in the example, we can achieve different ways of distributing an application by using newcog or new instructions at the object allocation sites. If newcog is used, a new distributed component is created, while when new is used, the created object (and its resource consumption) belongs to the current distributed component. As we have seen in the example, even for small programs, many different settings can be achieved by trying different combinations of newcog and new instructions. A careful inspection of our QAs can give very useful information to decide which setting is the most adequate for a specific deployment scenario, as we have seen in the examples of the previous section. However, we seek for a more systematic, and fully automated way to compare the different settings and evaluate their adequacy for a given deployment scenario. As all settings can be tried, this will provide us a way of finding the optimal setting for a distributed system.

We proceed in several steps: (1) we first consider in Section 6.1 *deployment constraints* that are provided externally by taking into account the deployment scenario (e.g., we might have limits on the number of distributed components that can be created). Such constraints might allow discarding settings that do not fulfill them, or inferring conditions on the input data that, if satisfied, ensure that the deployment constraints are met. (2) We then define in Section 6.2 a series of performance indicators that can be automatically computed from the information available in our QAs, and which are useful to estimate the quality of a setting. (3) Finally, as the *performance indicators* are given as functions on the input data sizes, Section 6.3 discusses practical issues to indicate when one configuration might be better than another one.

6.1. Deployment Constraints

Software is often developed for a range of deployment scenarios and indeed software performance can vary significantly depending on the target architecture. *Deployment constraints* can be used to express decisions that reflect the deployment scenario. For instance, as a cobox conceptually represents a processor, deployment constraints can be used to restrict the number of coboxes that can be created to the maximum number of processors available. In our proposal, such constraints are provided in the same language as the cost models we use in our framework. Also, they can refer to the resource consumption attributed to a set of objects $\mathcal{N}$. It can be the case that the deployment constraints are only *conditionally* preserved for certain input values.

Example 6.1. Let us consider a deployment scenario in which the number of processors available is 32. This deployment constraint is expressed in our framework by stating that the number of coboxes created by the program must be equal to or less than 32. It is stated using the cost model $\mathcal{M}^{\mathcal{C}}$ and the set of objects $\mathcal{N}$ are the coboxes, i.e., $\mathcal{N} = \{o_1, o_{12}, o_{13}\}$. In particular, $UB_{main}^{\mathcal{M}^{\mathcal{C}}}(n, m)|_{\{o_1,o_{12},o_{13}\}} \leq 32$. Setting 1 and setting 2 satisfy this constraint unconditionally for any possible values of n and m. However, setting 3 does not always satisfy the deployment constraint. Thus, we want to determine the conditions on the input arguments that guarantee that the program will run within the given constraints. The following condition is obtained $2 + n \leq 32$ that holds if $\varphi = \{n \leq 30\}$. The condition is given in terms of n due to the fact that the expression $UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})|_{\{o_1,o_{12},o_{13}\}}$ only depends on the value of the input argument n. ■

We denote by $\mathcal{L}_{\langle\mathcal{M},\mathcal{N}\rangle}$ a deployment constraint that restricts the resource consumption of the set of objects $\mathcal{N}$ w.r.t. the cost model $\mathcal{M}$. Note that $\mathcal{N}$ is a set of cost centers of the form $c(o)$ for the cost models $\mathcal{M}^I$, $\mathcal{M}^C$, and a set of pairs of the form $c(o_1, o_2), m$ for the cost model $\mathcal{M}^K$.

Definition 6.2. Given a program P with input arguments $\overline{x}$, a deployment constraint $\mathcal{L}_{\langle\mathcal{M},\mathcal{N}\rangle}$ for a cost model $\mathcal{M}$ and set of object names $\mathcal{N}$, we say that P conditionally satisfies $\mathcal{L}_{\langle\mathcal{M},\mathcal{N}\rangle}$, if there exist conditions φ on the input arguments $\overline{x}$ such that $\varphi \models UB_P^{\mathcal{M}}(\overline{x})|_{\mathcal{N}} \leq \mathcal{L}_{\langle\mathcal{M},\mathcal{N}\rangle}$.

In the previous definition we have focused on a single constraint for a particular cost model and a set of objects. In general, deployment constraints can refer to any resource of interest on any of the cost centers inferred by the analysis. This is because the scenario in which the application will be deployed can have more than one limitation. Having multiple constraints can fully discard a particular setting, but it can also result in multiple constraints on the input arguments. The constraints gathered from multiple resource limitations are conjoined to determine the global constraints on the input arguments (Definition 6.2 trivially extends to multiple constraints). If the conjunction results in an unsatisfiable set of constraints, we have that the setting is invalid for such deployment scenario.

Example 6.3. Let us suppose that, in addition to the constraints imposed in Example 6.1, we have a limitation on the number of instructions that each cobox can execute. If such limit is 500 instructions, we have that the coboxes inferred in the abstract configuration can execute at most 500 instructions. By using the UBs obtained for setting 3 in Example 5.7, we have that:

$$\begin{array}{lcl} \varphi_{o_1} & \equiv & 18 + 12 \cdot n \leq 500 \\ \varphi_{o_{12}} & \equiv & 6 \cdot n + 8 \cdot n \cdot m \leq 500 \\ \varphi_{o_{13}} & \equiv & 6 + 3 \cdot n \cdot m \leq 500 \end{array}$$

Thus, the conditions that warrant that the program will run within the given restrictions are:

$$\{n \leq 30\} \wedge \{6 \cdot n + 8 \cdot n \cdot m \leq 500\} \wedge \{6 + 3 \cdot n \cdot m \leq 500\}$$

■

6.2. Performance Indicators

We aim now at defining *performance indicators* that can be obtained from the information available in the QAs. Our goal is to use such indicators to compare different settings for the distributed system and find the optimal one (w.r.t. such indicators) for a particular deployment scenario. We note that there are aspects that might heavily influence the performance of the distributed system and that cannot be expressed using our framework. Therefore, we do not claim that the indicators that we define below are the unique criteria that matter in order to assess the performance. Instead, our indicators can be used in combination with other ones that currently cannot be obtained by state-of-the-art analyzers (see Section 9).

A *performance indicator* is defined as a function that evaluates into a number in the range $[0-1]$, such that the closer to one the better the performance. We start by defining the *distribution function*, which measures how much distributed the application is. It is defined as the relation between the number of coboxes that are created for this particular setting with respect to the maximum number of potential coboxes that could be created if all object instances were coboxes, i.e., the optimal setting from a distribution perspective in which we have as many coboxes as possible. In what follows, given a set of object names $\mathcal{N}$ we will use $coboxes(\mathcal{N})$ to refer to the elements in the subset of $\mathcal{N}$ that are cobox roots, $coboxes(\mathcal{N}) = \{o_\lambda \in \mathcal{N} \mid is\_root(o_\lambda)\}$.

Definition 6.4 (distribution function). Given a program P with input arguments $\overline{x}$, and its QAs, we define the *distribution level* for P as:

$$\mathcal{D}_P(\overline{x}) = \frac{UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})|_{coboxes(\mathcal{O})}}{UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})|_{\mathcal{O}}}$$

The distribution function is useful in the deployment process to know how close the application is to the maximum level of distribution that the program can reach (if all allocation sites are newcog instructions).

Example 6.5 (distribution function). The distribution function for setting 3 is computed as:

$$\mathcal{D}_{main}(n,m) = \frac{UB_P^{\mathcal{M}^{\mathcal{C}}}(n,m)|_{\{o_1,o_{12},o_{13}\}}}{UB_P^{\mathcal{M}^{\mathcal{C}}}(n,m)|_{\{o_1,o_{12},o_{13},o_{34}\}}} = \frac{2+n}{3+n}$$

The following table shows the distribution functions for all settings:

Setting 1	Setting 2	Setting 3
$\frac{2}{3+n}$	$\frac{3}{3+n}$	$\frac{2+n}{3+n}$

The deployment constraints in Section 6.1 can be used to discard any setting by imposing limits on the distribution level. ■

Another crucial aspect that can be observed using QAs is the level of *external* communications performed in the distributed system (i.e., calls to objects that belong to other coboxes). The motivation is that calls to other distributed components are potentially more expensive (as they require communications costs) and thus one wants to minimize them as much as possible. So as to evaluate this aspect, we use the *communication function*, which is defined as one minus the ratio between the number of communications that the program performs in the current setting, and the maximum number of communications when using a setting in which all objects are created as coboxes and thus every asynchronous call (on an object different from the one executing) is external.

Definition 6.6 (communication function). Given a program P with input arguments $\overline{x}$, and its QAs, we define the *communication level* for P as:

$$\mathcal{K}_P(\overline{x}) = 1 - \frac{UB_P^{\mathcal{M}^{\mathcal{K}}}(\overline{x})|_{comms(\mathcal{O}),methods(P)}}{UB_P^{\mathcal{M}^{\mathcal{K}}}(\overline{x})|_{allComms(\mathcal{O}),methods(P)}}$$

where $comms(\mathcal{O})$ is the set of pairs $(o_{\lambda_1}, o_{\lambda_2})$ such that o_{λ_1} and o_{λ_2} belong to different coboxes in the abstract configuration, $allComms(\mathcal{O})$ is defined as the set of all distinct pairs of objects, i.e., $\{(o_{\lambda_1}, o_{\lambda_2}) \in \mathcal{O} \times \mathcal{O} \mid o_{\lambda_1} \neq o_{\lambda_2}\}$, and $methods(P)$ is the set of methods defined in P.

Observe that, unlike the distribution function, which returns a greater value when more coboxes are created, the communication function will then return a smaller value when a larger number of coboxes are created (since more external communications are performed). Thus, there is a trade-off between these two performance indicators such that the optimal setting should take both indicators into account in the context of the specific scenario in which the distributed application will be deployed.

Example 6.7 (communication function). According to Definition 6.6, and using the UBs obtained in Example 5.14, the communication function for setting 3 is obtained as follows:

$$\mathcal{K}_{main}(n,m) = 1 - \frac{UB_P^{\mathcal{M}^{\mathcal{K}}}(\overline{x})|_{\{(o_1,o_{12}),(o_1,o_{13}),(o_{12},o_{13})\},methods(main)}}{UB_P^{\mathcal{M}^{\mathcal{K}}}(\overline{x})|_{allComms(\mathcal{O}),methods(main)}} = 1 - \frac{1+n+n\cdot m}{1+n+2\cdot n\cdot m}$$

Observe that the numerator expression accounts for all communications performed between objects that belong to different coboxes, while the denominator contains all possible pairs with different objects created in the program (calls in the same object are not added). The communication functions for our settings are:

Setting 1	Setting 2	Setting 3
$1 - \dfrac{n\cdot m}{1+n+2\cdot n\cdot m}$	$1 - \dfrac{n\cdot m+n\cdot m}{1+n+2\cdot n\cdot m}$	$1 - \dfrac{1+n+n\cdot m}{1+n+2\cdot n\cdot m}$

■

The third performance indicator that can be obtained using techniques based on resource analysis is the *balance level* of the distributed system. We consider that the system is *optimally* balanced when all its components execute the same number of instructions. In order to define this performance indicator, we introduce the *balance function* that makes use of the UBs on the number of instructions obtained using the cost model $\mathcal{M}^{\mathcal{I}}$ and the UBs on the number of objects in its QAs. The balance function measures the standard deviation of the number of instructions executed by each cobox. As a reminder, $s_N = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - avg(x))^2}$ is the deviation where $\{x_1, x_2, \ldots, x_N\}$ are the observed values of the sample items and $avg(x)$ is the mean value of these observations, while the denominator N stands for the size of the sample. To obtain the balance level, we consider that one observation is the number of instructions executed by each cobox. The number of observations of each cobox is multiplied by the number of instances identified for the abstract cobox to weight those abstract coboxes that might represent more that one concrete cobox. Finally, we want to measure the balance level by means of a number in the interval $[0, 1]$ as in the other indicators. For this purpose, we divide the standard deviation by the maximum dispersion of the coboxes from the average. The maximum dispersion for a data set is the expression $|x^+ - x^-|/2$, where x^+ and x^- represent the furthest (highest and lowest) values from the average of the data set (see, e.g., [ASY09]). Let us see the formal definition:

Definition 6.8 (balance function). Given a program P and input values $\overline{x}$ and its abstract configuration, we define the *balance deviation* of $P(\overline{x})$ as:

$$\sigma_P(\overline{x}) = \sqrt{\frac{\displaystyle\sum_{o_\lambda \in coboxes(\mathcal{O})} \left(\frac{UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})|_{\{o_\lambda\}}}{UB_P^{\mathcal{M}^{c}}(\overline{x})|_{\{o_\lambda\}}} - \frac{UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})|_{\mathcal{O}}}{UB_P^{\mathcal{M}^{c}}(\overline{x})|_{coboxes(\mathcal{O})}}\right)^2 * UB_P^{\mathcal{M}^{c}}(\overline{x})|_{\{o_\lambda\}}}{UB_P^{\mathcal{M}^{c}}(\overline{x})|_{coboxes(\mathcal{O})}}}$$

The *balance function* for P and $\overline{x}$ is defined as:

$$\mathcal{B}_P(\overline{x}) = 1 - \frac{\sigma_P(\overline{x})}{\left(\dfrac{UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})|_{\mathcal{O}}}{2}\right)}$$

Let us explain the details of the definition. The mean of the number of instructions executed by each cobox is obtained by accumulating the total number of instructions executed (obtained using the cost model

$\mathcal{M}^{\mathcal{I}}$) and dividing it by the number of coboxes (obtained using the cost model $\mathcal{M}^{\mathcal{C}}$ restricted to the set $\mathit{coboxes}(\mathcal{O})$). Then, the mean is captured by the expression:

$$\frac{UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})|_{\mathcal{O}}}{UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})|_{\mathit{coboxes}(\mathcal{O})}}$$

As regards the number of instructions executed by each cobox in the sample, we have to take into account that an abstract cobox might represent multiple concrete coboxes. Therefore, the number of instructions executed by an abstract cobox is accounting for the instructions executed by all coboxes it represents. The simplest solution that we adopt in order to define our performance indicator is to assume that the instructions are executed by such coboxes in an optimally balanced way (i.e., we divide the total number of instructions executed in the abstract cobox by the total number of coboxes that it represents). Thus, we divide the total number of instructions allocated to the abstract cobox o_λ, $UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})|_{\{o_\lambda\}}$, by the number of instances of the corresponding cobox $UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})|_{\{o_\lambda\}}$. Besides, as one abstract cobox might represent multiple concrete coboxes, we multiply the observation of each cobox by the UB on the number of instances, $UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})|_{\{o_\lambda\}}$. The values of the indicator must range in the interval $[0, 1]$. The minimum is 0, i.e., one cobox does not execute any instruction. The maximum is the total number of instructions executed by the program, i.e., all instructions are executed in only one cobox. Observe that the higher the balance deviation, the worse the balance function is. If the balance deviation is zero, that means that the coboxes are perfectly balanced, and therefore the balance function $\mathcal{B}_P(\overline{x})$ is 1.

Example 6.9 (balance function). For obtaining the balance function for setting 3, we first obtain the average on the number of instructions executed by each cobox. In such setting, the coboxes identified in the abstract configuration are $\{o_1, o_{12}, o_{13}\}$, thus the average is:

$$\mathit{avg}(UB) = \frac{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{I}}}(n,m)|_{\{o_1,o_{12},o_{13}\}}}{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{C}}}(n,m)|_{\{o_1,o_{12},o_{13}\}}} = \frac{24 + 18n + 11 \cdot n \cdot m}{2 + n}$$

We then generate the subexpressions associated to each cobox in $\{o_1, o_{12}, o_{13}\}$. The UB expressions for number of instructions and number of cobox instances are shown in the tables in Examples 5.7 and 5.9, respectively. They are used to obtain:

$$UBC_{o_1} = \frac{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{I}}}(n,m)|_{\{o_1\}}}{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{C}}}(n,m)|_{\{o_1\}}} = \frac{18 + 12 \cdot n}{1}$$

$$UBC_{o_{12}} = \frac{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{I}}}(n,m)|_{\{o_{12}\}}}{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{C}}}(n,m)|_{\{o_{12}\}}} = \frac{6 \cdot n + 8 \cdot n \cdot m}{n}$$

$$UBC_{o_{13}} = \frac{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{I}}}(n,m)|_{\{o_{13}\}}}{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{C}}}(n,m)|_{\{o_{13}\}}} = \frac{6 + 3 \cdot n \cdot m}{1}$$

The maximum dispersion of the coboxes is given by the expression:

$$\sigma_{\mathit{max}} = \frac{UB_{\mathit{main}}^{\mathcal{M}^{\mathcal{I}}}(n,m)|_{\{o_1,o_{12},o_{13}\}}}{2} = \frac{24 + 18n + 11 \cdot n \cdot m}{2}$$

By putting all together, the balance function is as follows:

$$\mathcal{B}_{\mathit{main}}(n,m) = 1 - \frac{\sqrt{\frac{(UBC_{o_1} - \mathit{avg}(UB))^2 * 1 + (UBC_{o_{12}} - \mathit{avg}(UB))^2 * n + (UBC_{o_{13}} - \mathit{avg}(UB))^2 * 1}{2+n}}}{\sigma_{\mathit{max}}}$$

■

The assumption made about the optimally balanced distribution within the component that an abstract cobox represents can be improved in several ways. First, by using a more accurate points-to analysis (e.g.,

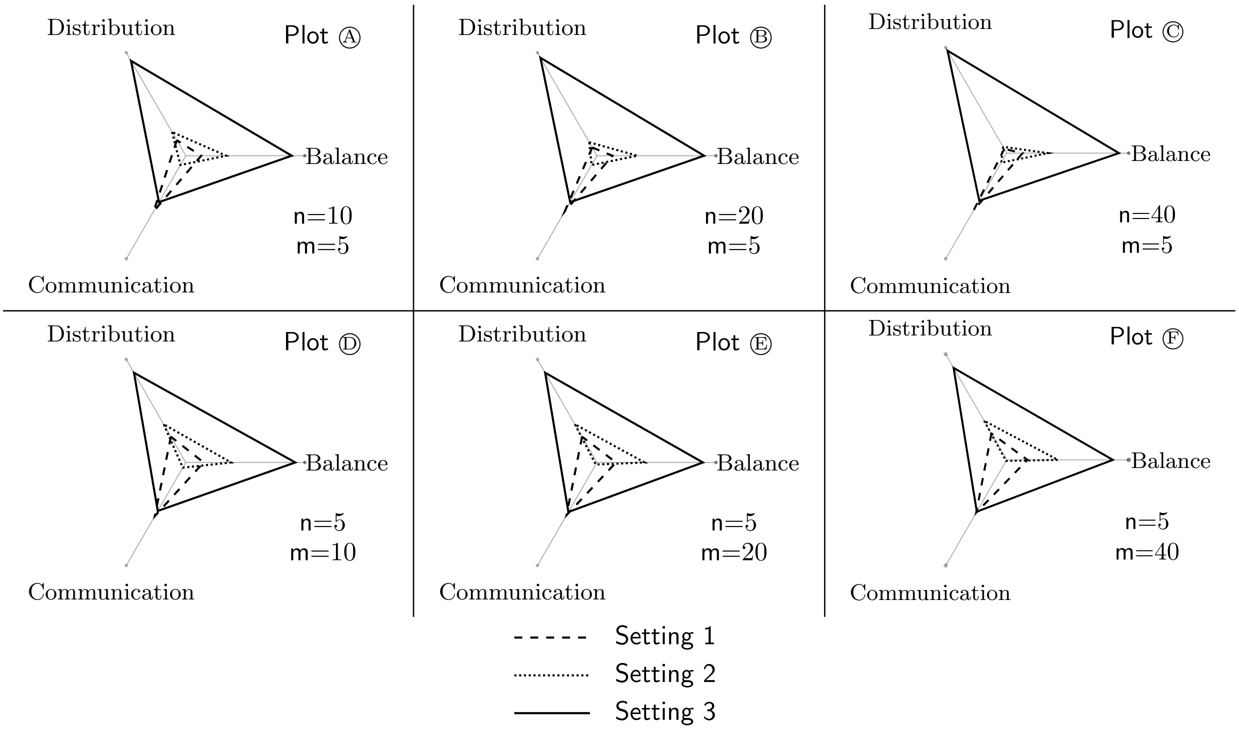


Fig. 11. Comparison for the runnnig example for different values of n and m

larger values of constant k), the loss of precision will be reduced. Still, over-approximation might be needed in some examples. Second, instead of assuming that the load is well balanced, one can make other assumptions, for instance, that the distribution is as bad as possible. The latter assumption can be realized by assuming that one concrete cobox executes all instructions and the remaining ones are idle (execute 0 instructions). Any other assumption in the middle can be made. Our well-balancedness assumptions works well for the examples we have tried, but other case studies may require different assumptions.

6.3. Overall Assessment of Setting

So far, we have probably discarded settings that do not satisfy deployment constraints, or we have constrained the input arguments to satisfy them, and we have computed performance indicators. The last step is to have an overall assessment of the considered settings and, then be able to compare them. We discuss three aspects relevant to such comparison: (1) evaluation for fixed input values, (2) plotting the performance functions, and (3) providing user-defined objective functions.

Fixed Input Arguments. The most trivial assessment is obtained by considering a range of fixed values for the input parameters. This is a very common situation when we are planning to deploy an application with an initial estimation on the number of potential customers or when we are interested in giving service to a concrete number of customers and, above this number, it is preferable to deny the requests because we cannot guarantee quality of service any longer. Since performance indicators are functions that evaluate to values in the range $[0-1]$, it is straightforward to instantiate them for particular input values.

Example 6.10 (fixed arguments). Let us study the distribution level, the communication level and the balance level of the running example for different pairs of the arguments n and m. Figure 11 shows graphically the results for different values of the input arguments. The plots have three axes, distribution, communication and balance, that correspond to the distribution function, the communication function and the balance function, respectively. The range shown for each axis is from 0 to 1 and the value obtained for each function

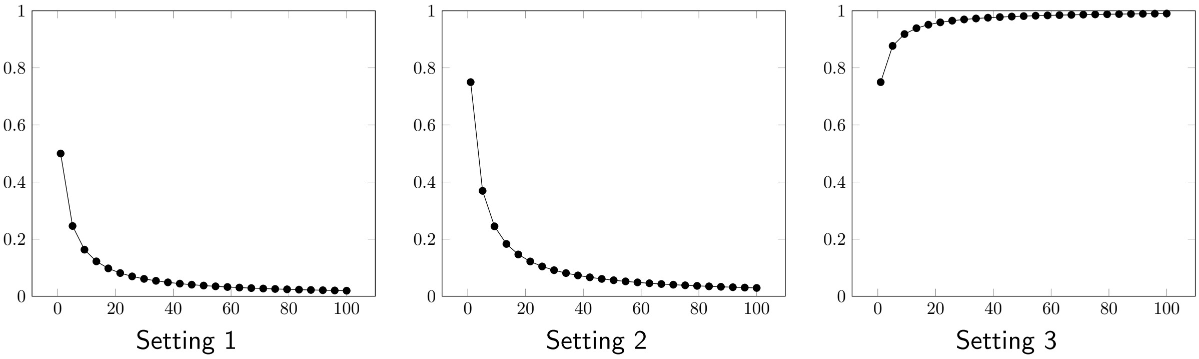


Fig. 12. Graphical representation of the distribution function for the running example

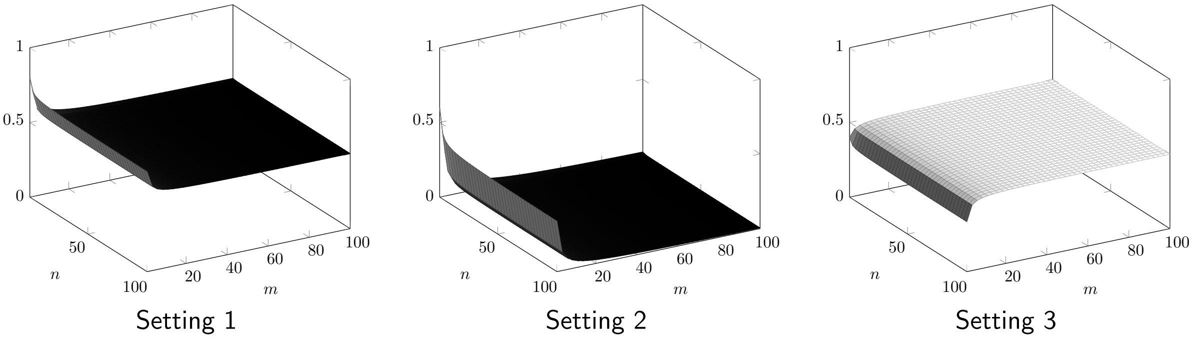


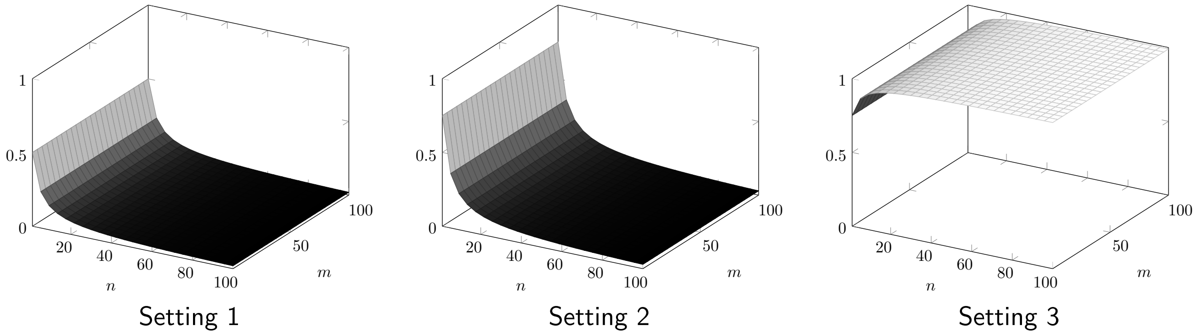
Fig. 13. Graphical representation of the communication function for the running example

is the result of the corresponding function for the selected input values. Besides, each plot shows three lines that correspond to the values obtained for setting 1, setting 2 and setting 3, shown in detail in Example 4.1. Such plots will be useful for comparing the behaviour of different settings for the given input arguments. Let us start by focusing on the plots that correspond to the pairs n=20, m=5 and n=5, m=20. Regarding distribution and balance, for both pairs it can be seen that setting 3 has higher values, which means that setting 3 has a better balance and distribution level than setting 1 and setting 2. As regards communications, setting 1, shows good results for both estimations. This is because most methods are executed by only one cobox. However, the behaviour of setting 3 is very similar in this point, showing that for both estimations, n=20, m=5 and n=5, m=20, setting 3 is the one that best fits with the deployment estimations.

As general conclusion, we can say that setting 1 and 2 will not distribute the work properly and consequently will behave worse when the number of clients (n) is increased. This aspect can be observed in plots Ⓐ, Ⓑ and Ⓒ, that show how the distribution level gets worse when n is increased. On the contrary, the number of requests does not affect them, plots Ⓓ, Ⓔ and Ⓕ are very similar for all values of m. This means that settings 1 and 2 could be valid for deployment scenarios with an estimation of a low number of clients, independently of the number of requests generated by each client. Clearly, setting 3 is the one that behaves better for all scenarios studied, and different values of n and m do not affect the behaviour of the resulting configuration. Note that one important point, in setting 3, is that the number of processors required for solving the requests depends on the value of n. Thus, depending on the number of processors available, this configuration could be invalid for a large number of clients. Such kind of restrictions can be handled by using the deployment constraints imposed in Example 6.3.

■

Non-fixed Input Arguments. The problem of fixing the input arguments is well-known, one can miss the anomalous behaviour (in our case related to the program's resource consumption). Therefore, it is interesting to observe how the resource consumption evolves with larger values of the input arguments. One can also



Setting 1 Setting 2 Setting 3

Fig. 14. Graphical representation of the balance function for the running example

focus on a range of interest. As performance indicators are functions, it is possible to visualize their behaviour by representing them graphically, provided the functions do not have too many arguments.

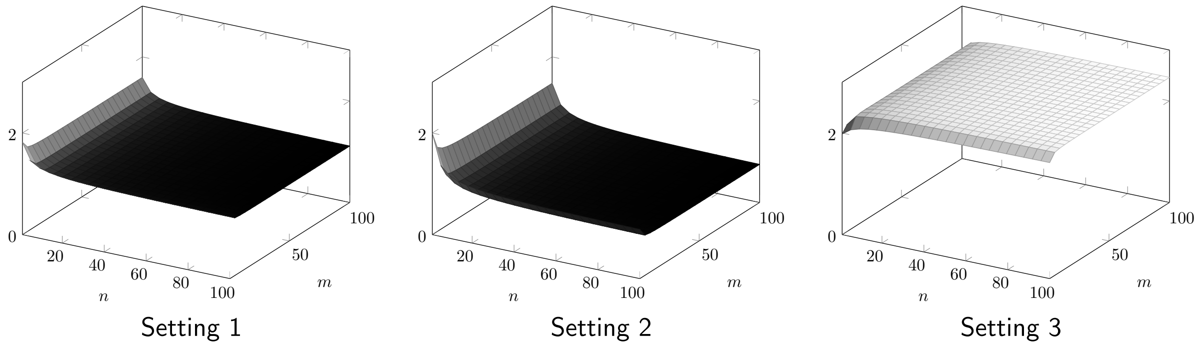
Example 6.11 (non-fixed arguments). Figure 12 shows how the distribution function for the running example (see Example 6.5) evolves for all settings of interest. This graphical representation clearly shows that the distribution levels of setting 1 and setting 2 get worse when the value of n is increased. However, the distribution level of setting 3 gets better when the value of n is growing.
In Figure 13 we can observe the behaviour of the communication function with respect to n and m. Figure 13 confirms that the communication level of setting 1 is the best one, but it also shows that increasing the value of n deteriorates the performance of this setting. On the contrary, the behaviour of setting 3 is quite stable for all possible values of n.
The balance function for all settings is shown in Figure 14. As it was explained in Example 6.10, the performance of settings 1 and 2 gets worse when the value of n increases. Once again we confirm that setting 3 is the most stable of all settings proposed for the program and it features a good trade-off between all features, distribution, communications and nodes balance. ■

Comparison of Different Settings. In order to find the optimal setting for a distributed system, we should be able to: (1) generate all possible settings automatically, (2) generate performance indicators for each of them and (3) be able to compare such indicators for the different settings. As regards (1), it should be noted that some settings should be discarded when generating all possible combinations of new and newcog instructions. As noted in [FMA13] by grouping objects in coboxes one can introduce deadlocks. Therefore, each candidate configuration should be checked for deadlock freeness. Apart from this issue, the generation of all possible settings does not pose any relevant problem. The second aspect has been discussed in the previous section. So, it remains to be seen how to automatically compare different settings.

In principle, since performance indicators are functions, point (3) consists in comparing the corresponding functions. There are several points to take into consideration here:

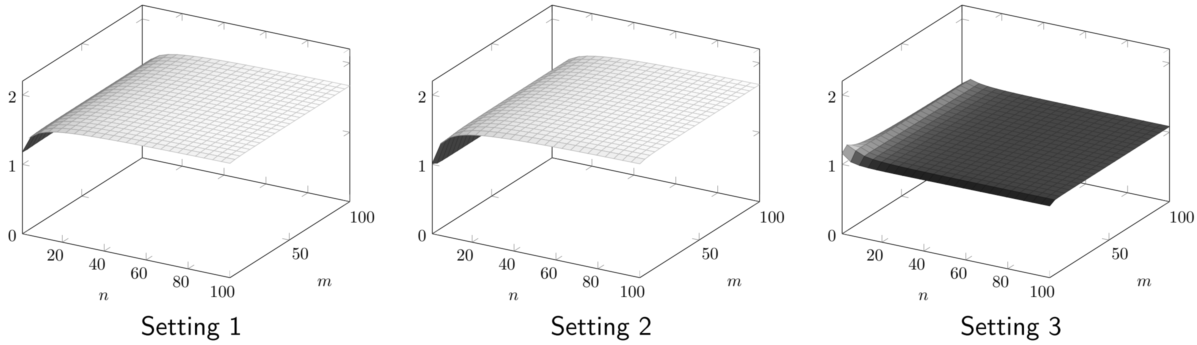
- Function comparison is undecidable. We refer to [AAG10] for a practical method for function comparison. There is no guarantee that we will be able to have a result from the comparison of the functions. Also, the work in [AAG10] does not admit square roots in functions such as the ones we have in the balance function. If function comparison cannot be performed, we can compare the settings by using fixed input arguments as discussed above.
- When one setting outperforms all others (according to all considered performance indicators) the result is clear. But it might happen that it outperforms other settings only for some indicators. In such case, one can define *target functions*, $\mathcal{T}(\overline{x})$, that weight the importance given to each performance indicator.

Target functions are functions that combine, by taking into account to the target deployment scenario, the performance indicators defined in Section 6.2. Essentially, they weight the indicators depending on the characteristics of the scenario. Similarly to the performance indicators, target functions are also expressed in terms of the input arguments of the program and return a value that could help to determine the quality of each setting for the target scenario. The simplest target function is the addition of all performance indicators, which gives the same weight to all indicators. However, such function could not be so good for



Setting 1 Setting 2 Setting 3

Fig. 15. Graphical representation of the target function $\mathcal{T}_{main}(n, m)$ for the running example



Setting 1 Setting 2 Setting 3

Fig. 16. Graphical representation of the target function $\mathcal{T}'_{main}(n, m)$ for the running example

a cloud computing scenario where connections are costly if computers are located in different countries. On the contrary, a scenario where all computers are connected through a fast local area network and the number of processors is limited, is well captured by using a function that gives higher weight to the distribution and less weight to communications. Let us see two target functions by means of an example.

Example 6.12. Let us start the example with a target functions that weights equally the performance indicators obtained in Example 6.5, Example 6.7 and Example 6.9:

$$\mathcal{T}_{main}(n, m) = \mathcal{D}_{main}(n, m) + \mathcal{K}_{main}(n, m) + \mathcal{B}_{main}(n, m)$$

Figure 15 shows the graphical representation of $\mathcal{T}_{main}(n, m)$. It can be observed that setting 3 behaves better than setting 1 and setting 2, as expected from the results obtained in the previous examples. Setting 3 shows a good behaviour when the values of n and m are increased because the program is well balanced and well distributed, and the number of communications does not degrade its performance.

Let us now consider a deployment scenario with some peculiar characteristics. Assume that communications are done by a fast communication channel and the number of processors is low. The latter requires that the distribution level is as low as possible, i.e. high levels of distribution should be mapped into low values in the target function. A target function that could represent such scenario is:

$$\mathcal{T}'_{main}(n, m) = (1 - \mathcal{D}_{main}(n, m)) + 0.2 * \mathcal{K}_{main}(n, m) + \mathcal{B}_{main}(n, m)$$

The graphical representation of $\mathcal{T}'_{main}(n, m)$ in Figure 16 indicates that the performance obtained for setting 1 and setting 2 is much better than that of setting 3. The main reason is that the unlimited number of coboxes created by setting 3 results in lower values of $\mathcal{T}'_{main}(n, m)$. ■

Benchmark	**loc**	$\#_{\mathsf{P}}$	$\#_{\mathsf{A}}$	$\mathbf{T}_{\mathcal{C}}$	$\mathbf{T}_{\mathcal{K}}$	$\mathbf{T}_{\mathcal{I}}$	$\mathbf{C}_{\mathcal{C}}$	$\mathbf{C}_{\mathcal{K}}$	$\mathbf{C}_{\mathcal{I}}$
Running	78	2	5	0.05	0.06	0.06	5/1/5	11/2/10	31/2/16
BBuffer	105	5	4	0.14	0.18	0.19	5/2/4	10/4/7	52/5/21
MailServer	115	3	4	0.10	0.15	0.18	4/1/4	16/3/8	59/5/20
Chat	302	4	10	0.11	0.13	0.16	10/1/10	38/2/34	96/4/46
BookShop	353	6	5	0.57	0.59	1.24	5/1/5	10/1/10	134/19/24
PeerToPeer	240	5	9	1.60	11.57	13.09	8/1/9	244/21/252	2642/225/1100
TradingSystem	1340	15	23	110.31	721.34	746.15	27/2/23	354/9/325	560390/18190/309705

Table 1. Experimental evaluation results (times in seconds)

7. Experimental Evaluation

We have implemented our analysis in SACO `http://costa.ls.fi.upm.es/SACO`, a Static Analyzer for Concurrent Objects, and we have applied it to some classical examples of distributed based systems: BBuffer, the typical bounded-buffer for communicating several producers and consumers; MailServer, which models a mail server distributed system with multiple clients; Chat, modelling a client-server chat application; BookShop, which implements a web shop client-server application; and the RunningExample shown in Figure 2. In addition, we have studied two case studies: an academic case study, PeerToPeer, which represents a peer-to-peer network formed by a set of interconnected peers which make some files available to other peers; and TradingSystem, an industrial case study that models a supermarket sales handling system. The source code for the benchmarks can be downloaded from the SACO web site. Due to some limitations of the underlying resource analysis that are not related to our method, we had to slightly modify the programs by changing the structure of some loops, and had to add some size relations that the analysis could not infer (see [AAG11a] for more details about the class invariants needed).

7.1. Generation of Quantified Abstractions

We have computed the UB expressions for all programs for the cost models $\mathcal{M}^I$ (Definition 3.4), $\mathcal{M}^C$ (Definition 5.8) and $\mathcal{M}^K$ (Definition 5.13). During the analysis, the points-to graph (Definition 5.2) of the program has also been inferred, since the points-to information is needed to obtain the UBs with cost centers for the different objects. The experiments have been performed on an Intel Core i7 at 3.4GHz with 4GB of RAM, running Ubuntu Server 12.04. Points-to analysis has been performed with $k = 2$, i.e., the maximum length of object names (see Section 3) is two. Table 1 shows the efficiency of the analysis. Let us describe the figures in detail. Columns Benchmark, loc and $\#_{\mathsf{P}}$ show, respectively, the studied benchmark name, the number of lines of ABS code, and the number of input parameters. Column $\#_{\mathsf{A}}$ displays the number of allocation sites found in the program. Columns $\mathbf{T}_{\mathcal{C}}$, $\mathbf{T}_{\mathcal{K}}$ and $\mathbf{T}_{\mathcal{I}}$ show the time taken to compute three different cost analyses of the program, using the cost models $\mathcal{M}^C$, $\mathcal{M}^K$ and $\mathcal{M}^I$, respectively. Finally, columns $\mathbf{C}_{\mathcal{C}}$, $\mathbf{C}_{\mathcal{K}}$ and $\mathbf{C}_{\mathcal{I}}$ aim at showing the complexity of the UB expressions inferred by the resource analysis. The information shown in such columns has the format A/B/C, where A is the number of arithmetic symbols that appear in the UB, B is the number of occurrences of the input arguments in the UB, and C is the number of object names in the UB expression. The figures in $\mathbf{T}_{\mathcal{C}}$, $\mathbf{T}_{\mathcal{K}}$ and $\mathbf{T}_{\mathcal{I}}$ show that the fastest cost model for all benchmarks is $\mathcal{M}^{\mathcal{C}}$, whereas $\mathcal{M}^{\mathcal{I}}$ is the most costly one. This is explained by the fact that using $\mathcal{M}^{\mathcal{C}}$ the analysis will often find loops with zero cost (those that do not contain new or newcog instructions) for which the analyzer does not need to perform any solving. This aspect is especially remarkable in PeerToPeer and TradingSystem, which contain complex loops whose analysis is expensive, but as the allocation sites appear outside these loops, they do not add any cost to the computation of $\mathcal{M}^{\mathcal{C}}$. Columns $\mathbf{C}_{\mathcal{C}}$, $\mathbf{C}_{\mathcal{K}}$ and $\mathbf{C}_{\mathcal{I}}$ show that the time required to infer the UB increases significantly with the complexity of the expression. The size of the UB for TradingSystem is especially large, not only because of the number loops found in the program, but also because of the number of object names needed to keep track of the object on which each instruction is executed (see the third value in $\mathbf{C}_{\mathcal{I}}$ that is larger than 300 thousand names).

The result of the analysis for each benchmark is an upper bound for each of the cost models used. It must be stressed that, since the benchmarks considered are non trivial, such UBs are expressed in terms of

Benchmark	$\#_{\mathsf{S}}$	$\#_{\mathsf{CP}}$	$\mathbf{T}_{\mathsf{ES}}$	$\#_{\mathsf{CP/sec}}$	$\#_{\mathsf{E}}$	$\mathbf{T}_{\mathsf{T}}$
Running	16	100 200	17.48	5 732.26	1 603 200	1.687
Bounded Buffer	8	120 000	27.60	4 347.82	960 000	0.968
MailServer	8	102 000	24.64	4 139.61	816 000	0.817
Chat	512	12 000	9.87	1 215.31	6 144 000	7.295
BookShop	16	101 088	131.98	765.92	1 617 408	1.770
PeerToPeer	256	4 992	127.42	39.17	1 277 952	1.501
TradingSystem	64	1 024	254.03	0.06	256	1.010

Table 2. Experimental evaluation results (times in seconds)

(a subset of) the input arguments and using the object names identified in the points-to analysis to define cost centers. Observe that some input arguments may not appear in the UB expressions, as the configuration of the system may depend on a subset of the input parameters only. This is often the case for the cost model $\mathcal{M}^{\mathcal{C}}$ and can also be observed for $\mathcal{M}^{\mathcal{K}}$, where the number of occurrences of the input parameters in the UB expression is indeed smaller than the number of input parameters (see e.g. the second value in columns $\mathbf{C}_{\mathcal{C}}$, and $\mathbf{C}_{\mathcal{K}}$ of BookShop, or $\mathbf{C}_{\mathcal{K}}$ of TradingSystem).

7.2. Application for Finding Optimal Configurations

We now aim at finding the optimal configuration for our benchmark programs. We proceed in the next steps:

(1) **Settings generation**: Using the points-to information, we generate all possible settings, by creating all combinations of new and newcog instructions for the allocation sites found by the points-to analysis.

(2) **Performance indicators**: For each setting generated in (1), denoted by S_i, and using the UBs inferred in Section 7.1, we obtain the performance indicators, $\mathcal{D}_{\mathsf{S}_i}$, $\mathcal{K}_{\mathsf{S}_i}$ and $\mathcal{B}_{\mathsf{S}_i}$ (see their definitions in Section 6.3).

(3) **Input values**: We fix a range of concrete values for each input parameter and generate all combinations of the values for the parameters. We denote each combination of concrete values as P_j.

(4) **Performance indicators evaluation**: We evaluate the performance indicators generated in (2), for each parameter combination generated in (3). We use the expressions $\mathcal{D}_{\mathsf{S}_i}(\mathsf{P}_j)$, $\mathcal{K}_{\mathsf{S}_i}(\mathsf{P}_j)$, $\mathcal{B}_{\mathsf{S}_i}(\mathsf{P}_j)$ to identify the evaluation of the setting S_i for the concrete input parameters P_j.

(5) **Settings performance**: At this point, for each setting S_i generated at (1), we have three performance indicators evaluated for multiple input arguments (generated in (3)). Thus, for each setting S_i, we calculate the average of the performance indicators evaluated in (4). We denote the average of the performance indicators by $avg(\mathcal{D}_{\mathsf{S}_i})$, $avg(\mathcal{K}_{\mathsf{S}_i})$, $avg(\mathcal{B}_{\mathsf{S}_i})$.

(6) **Target Function**: Finally, we use the target function $\mathcal{T}_{\mathsf{S}_i} = avg(\mathcal{D}_{\mathsf{S}_i}) + avg(\mathcal{K}_{\mathsf{S}_i}) + avg(\mathcal{B}_{\mathsf{S}_i})$ to compare and rank the efficiency of all evaluated settings.

Table 2 shows the results of the application of the steps described above. Column $\#_{\mathsf{S}}$ shows the number of settings generated in step (1). Column $\#_{\mathsf{CP}}$ shows the number of combinations of the input parameter values on which each setting will be evaluated (step (3)). $\mathbf{T}_{\mathsf{ES}}$ shows the average time taken by the evaluation of the performance indicators (for one setting) for all input parameters values (steps (2), (3), (4)). Column $\#_{\mathsf{CP/sec}}$ shows the number of evaluations performed in one second. Column $\#_{\mathsf{E}}$ shows the total number of evaluations performed for each benchmark. Note that Chat requires more than 6 million evaluations since there are 512 settings that are tried out on 12000 combinations of the input parameters. Finally, column $\mathbf{T}_{\mathsf{T}}$ shows the time taken to perform the average described in step (5) and the application of the target function to sort the settings, as described in step (6).

Observe that the evaluation described in these experiments evaluates all possible settings for a large number of combinations. One efficient improvement can be achieved by reducing the number of settings generated in Step (1), e.g., by fixing some allocation sites of interest to new or newcog. The number of combinations of input values generated in Step (3) can also be reduced if there is additional information of the actual system being configured given by means of constraints. Another interesting aspect is that the evaluation of a setting for a specific set of input values is independent from evaluating other settings, or other input values of the same benchmark program, and thus the process can be parallelized.

We can observe that the number of evaluations per second ($\#_{\mathsf{CP/sec}}$) directly depends on the complexity of the UB expressions and the evaluation significantly varies for different example benchmarks, ranging from 4347 evaluations per second in BoundedBuffer to only 765 in BookShop. The same pattern is observed in the PeerToPeer and TradingSystem case studies, in which the size of the UB expressions is decisive in the time taken in the evaluation of performance indicators. For PeerToPeer, it results in a low number of evaluations per second (39) and, in the case of TradingSystem, the evaluation of the performance indicators for one concrete set of values of the input parameters takes around 15.8 seconds. In Section 7.3 we will see how we can handle such issue in practice. One positive aspect is that, once the performance indicators are obtained, the time to compute the target function ($\mathbf{T}_{\mathsf{T}}$) is insignificant. This means that several target functions can be efficiently computed from the results obtained for the performance indicators.

7.3. Case Studies

Once we have seen that the systematic evaluation of all settings for the benchmarks is feasible and relatively efficient, let us study in detail the more complex case studies, PeerToPeer and TradingSystem. We first focus on the PeerToPeer, a well-known case study for distributed systems. The main classes that compose this system are: OurTopology, which represents the topology of the peer-to-peer system; Node, that models the nodes that share the files; and Database, which represents the files database used by the nodes. A node can provide its catalog of files to its neighbours by accessing the database, and can obtain the list of its neighbours from the topology. Nodes can also request and transfer files, package by package, to its neighbours. The plots in Figure 17 display the values of the performance indicators and the target function described in step (6) for all possible settings. The vertical axis of plots 1, 2, 3, 4 shows, respectively, the values obtained for $avg(\mathcal{D}_{\mathsf{S}_i})$, $avg(\mathcal{K}_{\mathsf{S}_i})$, $avg(\mathcal{B}_{\mathsf{S}_i})$ and the value of the target function, $\mathcal{T}_{\mathsf{S}_i}$. Settings are sorted by the value $\mathcal{T}_{\mathsf{S}_i}$ in descending order. Plot 4 shows that the target function ranges from 1.55 to 0.55, confirming, as expected, that the setting selection directly affects the behaviour of the program in terms of distribution, communication and balance. Another interesting aspect is that multiple settings have the same target function value. This is due to the fact that having new or newcog in some allocation sites dominates the general behaviour of the whole system, independently of the remaining allocation sites.

Now, let us focus on the first five settings (plot 5 in Figure 17). Settings 1 and 2 clearly show the trade-off between distribution and communication. In setting 1 all nodes are executing in just one cobox resulting in a high communication but a low distribution level. On the contrary, Setting 2 is fully distributed (all allocation sites are newcog), resulting in a excellent distribution, but a poor communication level. Both settings should be discarded for the deployment: Setting 1 because in a peer-to-peer environment nodes must execute in separate coboxes, and Setting 2 because the high number of communications could affect the performance of the application. In these experiments we used a simple target function. Nonetheless, this aspect can be considered by defining a target function that better reflects the structure of a peer-to-peer application. Therefore, this benchmark confirms the importance of fixing some allocation sites to concrete instructions, as well as the relevance of defining an appropriate target function.

Regarding setting 3, its performance indicator shows a good trade-off between balance, communication and distribution. Such setting generates a cobox that takes care of the topology, another cobox for handling the databases, and one cobox per node. Definitely, this setting generates an appropriate configuration for a peer-to-peer system. Settings 4 and 5 show a similar behaviour to setting 2, so they should also be discarded.

The TradingSystem case study models a supermarket cash desk line. It includes the processes at a single cash desk (like scanning products using a bar code scanner, paying by cash or by credit card); it also handles bill printing, as well as other administrative tasks. In the TradingSystem, 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 time to analyze the TradingSystem is larger than that of the other benchmarks. This is an expected result, due to the complexity of the UB expressions shown in Table 1 and the size of the application. Furthermore, it contains 22 allocation sites, whose combinations result in more than 4 million possible settings. By inspecting the program, we have detected that several objects must be created in separate coboxes (i.e., using newcog) because they represent independent devices with their own processors, such as a printer, a bar code scanner or a card reader. Besides, two allocation sites must be created using new because they are part of the same physical device as the creator object. Therefore, there are 9 allocation sites left to be created using new or newcog, resulting in the evaluation of 512 settings. Since the evaluation of all settings for multiple combinations would take too much time, we have decided to divide

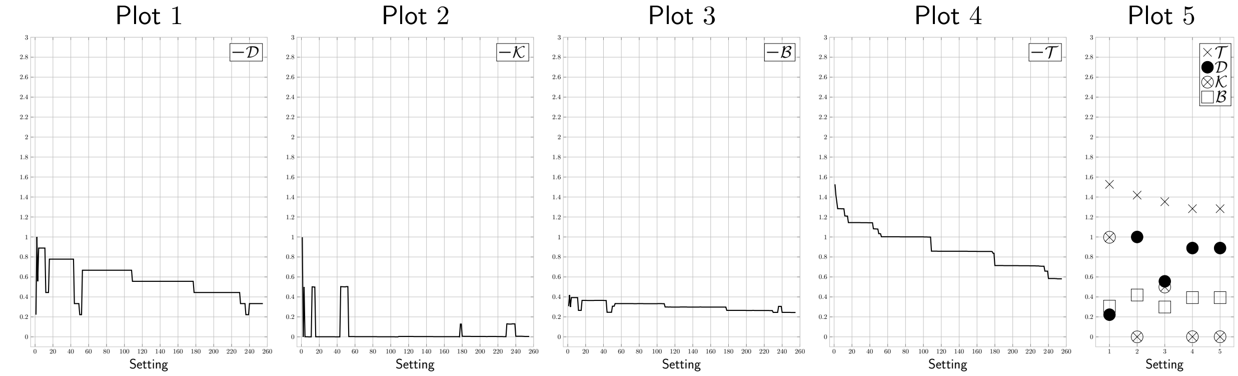


Fig. 17. PeerToPeer case study evaluation (256 settings sorted by $\mathcal{T}$)

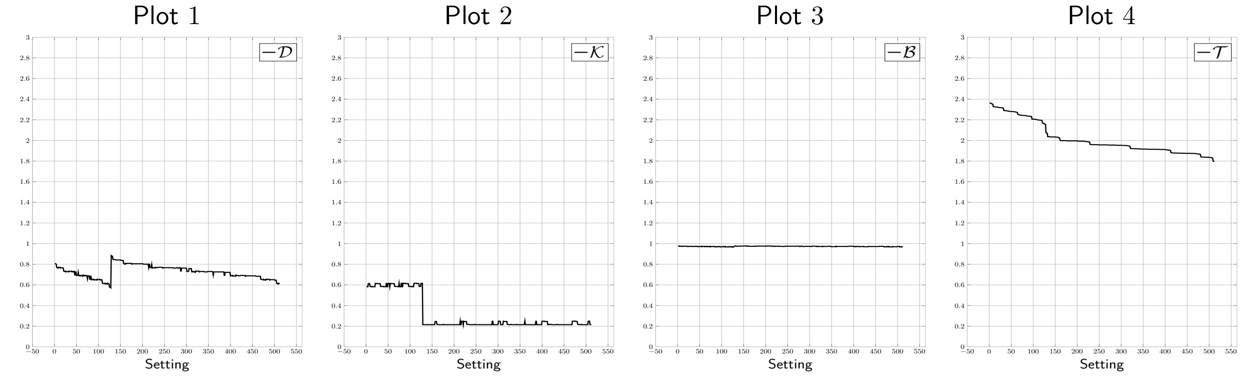


Fig. 18. TradingSystem case study evaluation (512 settings sorted by $\mathcal{T}$)

the analysis in two different steps: (1) evaluate all possible settings for one concrete evaluation of the input arguments, and, (2) using the information obtained from (1), set to newcog or new those allocation sites that produce a significant change in the values of $\mathcal{T}$. The plots in Figure 18 show the evaluation of all 512 settings for TradingSystem. At certain point the value of $\mathcal{K}$ falls down drastically, resulting in a significant reduction in the value of $\mathcal{T}$. By inspecting these settings, we observe that this is because the objects CashDeskPCImpl and CashDeskInstallationImpl are highly cooperative. Thus, the settings that separate both objects in different coboxes have a lower value for $\mathcal{K}$, and consequently for $\mathcal{T}$. The same reasoning is also applicable to objects CashBoxPCImpl and CashDeskInstallationImpl. Therefore, these objects have been fixed to be created using

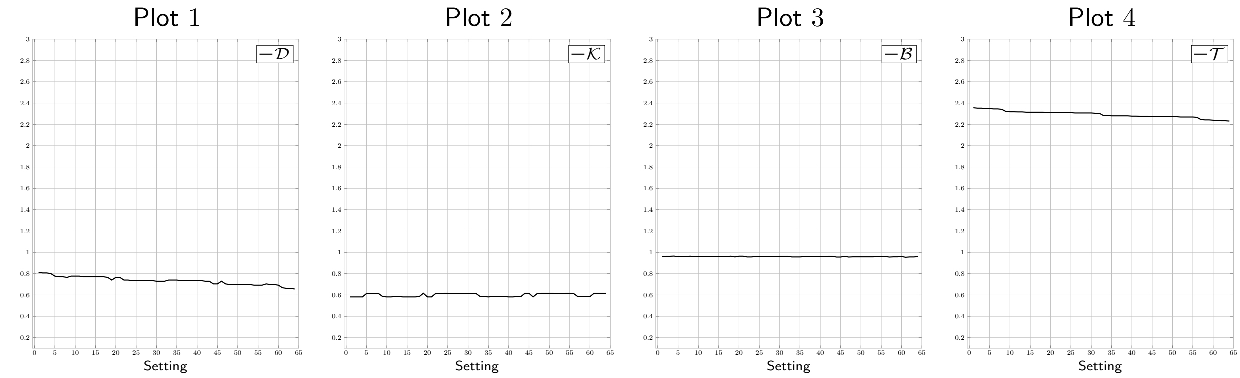


Fig. 19. TradingSystem case study evaluation (64 settings sorted by $\mathcal{T}$)

new. The number of possible settings is now 64. We have recomputed the performance indicators for 16 combinations of the input arguments for these settings. Plot 4 in Figure 19 shows that $\mathcal{T}$ ranges from 2.2 and 2.4, which essentially reflects that CashDeskPCImpl and CashBoxPCImpl are decisive in the configuration of the system. It can also be observed in plot 3 of Figure 19 that the balance level is almost constant in this case study. As several objects must be created with newcog since they represent independent devices, and those objects are indeed well distributed, this performance indicator is not relevant in this case.

Summarizing, we argue that it is feasible to apply the approach proposed in this article to models of real systems, and that performance indicators are helpful to guide the deployment process. Some crucial aspects for the efficiency and usefulness of our approach are: first, the selection of precise ranges for the input arguments; second, filtering some non-possible configurations by setting some allocation sites to new or newcog; and finally, defining a precise target function that faithfully describes the expected behaviour of the system.

8. Related work

The analysis presented in the article is based on two existing analyses, resource and points-to analyses, whose integration allows defining the concept of quantified abstract configuration. We first review related work on these two techniques. As regards resource analysis, our work builds upon an existing framework for cost analysis [AAG07, AAG08, AAG11a, AFG13]. Such framework has been defined in several steps. First, it was formalized how to generate cost recurrence equations from a Java-like imperative language in [AAG07, AAG12b]. This approach applies directly to the sequential fragment of our language. Then, a method to obtain upper bounds from cost relations was defined in [AAG08, AAG11b]. This method is language independent and thus is used to solve the cost relations that we produce from ABS programs without requiring any extension.

Later work [AAG11a, APC12] focuses on generating cost recurrence equations from ABS programs. This requires non-trivial extensions to treat the task interleavings that may occur in the execution of ABS programs. In particular, when a task suspends due to the use of an `await` instruction, another task can take the processor and modify the object fields. The approach in [AAG11a, APC12] proposes to lose the values of fields at any potential release point. A more accurate solution has been proposed later to avoid losing the values of fields [AFG13] when the tasks whose execution may interleave either do not modify the involved fields or if they do so, they modify them a finite number of times. These techniques (both the initial and the more accurate one) constitute the resource analysis framework that our work relies upon, and they have been adopted directly for the generation of the upper bounds along the article.

As regards the points-to analysis, as already mentioned, we are using the abstraction proposed by Milanova et al. [MRR05] that fits perfectly with the concept of concurrent object used in our language. In essence, as each object is a concurrency unit, it is fundamental for the precision of the analysis to distinguish information at the object level. The resource analysis framework defined in [AAG11a, APC12] is already object-sensitive. However, the difference is that it does not use the notion of configuration and, thus, it does not keep track of the cobox to which object belongs. This piece of information is essential to define the notion of quantified abstraction and to infer meaningful information on the distributed system.

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 π-calculus and BioAmbients [Fer01, RL05]. In [PHW05], a static analysis based on probabilistic abstract interpretation is proposed to deal with distributed systems. The approach is quantitative in the sense that it obtains an approximation of the probability that some property of interest is transmitted through a distributed network within a given interval of time. As the previously mentioned contributions, it is applied to a high-level experimental language, pcKLAIM, a restricted version of KLAIM (Kernel Language for Agents Interaction and Mobility) with no higher order features and single parameter passing. It does not deal with resource analysis in general (such as memory, number of instructions, etc.), but only communication issues.

The techniques that we use in this article are mainly static, i.e., we obtain the resource consumption without executing the program, by only inspecting the program code. The results of static analysis are *sound* for any execution of the program. This is because our analysis has considered the resource consumption of all feasible paths of execution (and interleavings) and has not left any unchecked behavior. The analysis returns the worst-case cost of the information obtained from all paths. In contrast, dynamic analysis (a.k.a. profiling) gathers information from running the system. In general, the collected data is accurate of system execution as long as the overhead of the measurement has not influenced the results. Profiling is limited, however, to the inspection of behavior that can be made by running the system on a selection of input. This limitation means that profiling is useful in circumstances where a sampling of behavior is sufficient. It is clearly not well suited to ensure an optimal behavior in a system when only one execution in a million can lead to a large resource consumption. In our experiments we have somehow combined static and dynamic techniques in the sense that the upper bounds that we obtain statically are evaluated for particular input values. It should be clear that our results cannot be obtained by profiling the program, since the upper bounds that are evaluated are already safe approximations of the overall cost. Thus, we do not have the problem of missing the anomalous behaviour of the system in the selected input data as in profiling.

To the best of our knowledge, this paper is the first static approach that presents a quantitative abstraction of a distributed system for a real language and experimentally evaluates it on a prototype. We argue that our work is a first crucial step towards automatically inferring optimal deployment configurations of distributed systems.

9. 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. We have seen that *performance indicators* can be obtained from our quantified abstract configurations. Our indicators estimate the distribution level of the system, the amount of communication among its distributed components and the load balancing. These indicators can be useful to determine that one setting has a better performance than another one and, if all possible settings are tried, to find the optimal one (according to our indicators). We do not claim that our performance indicators are the only ones that must be taken into account, but rather they are useful indicators that can be obtained from today's resource analyzers. Future work will focus on defining complementary indicators as discussed below.

Currently, our upper bounds on the number of execution steps do not take into account the fact that tasks can run in parallel in different distributed components. Thus, instead of accumulating the steps performed in each component, in future work, we want to develop new techniques to infer the maximum amount of steps that are performed when both components run in parallel. This is a first step towards the inference of the runtime of the distributed system. In particular, it can happen that the load on the distributed nodes is well balanced, however, the execution performed in the different components is serial, i.e., they will not happen in parallel. Thus, one does not benefit from distributing these components in different machines. The global runtime will be another performance indicator that can be used to choose among different possible settings.

Besides estimating runtime, we also plan to perform bandwith analysis that requires to approximate the sizes of the data in asynchronous calls. This is a non-trivial problem that will require the development of sophisticated size analyses. Again, bandwith analysis will provide another relevant performance indicator that can be used in our framework in the same way as the communication level it is currently used.

Acknowledgements. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

References

[AAG07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[AAG10] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6234 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

[AAG11a] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, December 2011.

[AAG12a] 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*, pages 151–154. ACM Press, 2012.

[AAG12b] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.

[AAG08] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proc. of SAS 2008*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[AAG11b] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[ACP13] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Quantified Abstractions of Distributed Systems. In *Proc. of iFM'13*, volume 7940 of *LNCS*, pages 285–300. Springer, 2013.

[AFG13] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*, LNCS 8172, pages 349–364. Springer, October 2013.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[APC12] E. Albert, P.Arenas, J. Correas, M. Gómez-Zamalloa, S. Genaim, G. Puebla, and G. Román-Díez. Object-sensitive cost analysis for concurrent objects. http://costa.ls.fi.upm.es/papers/costa/AlbertACGGPRtr.pdf, 2012.

[ASY09] M. F. Al-Saleh and A. E. Yousif. Properties of the standard deviation that are rarely mentioned in classrooms. *Austrian Journal of Statistics*, 38(3):193–202, 2009.

[AVW96] J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[BCG07] M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based norms. *TOPLAS*, 29(2):Art. 10, 2007.

[BBS13] J. Bjørk, F. S. de Boer, E. Broch Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.

[BYV09] 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.

[Car93] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[BGS12] F. S. de Boer, I. Grabe, and M. Steffen. Termination detection for active objects. *J. Log. Algebr. Program.*, 81(4):541–557, 2012.

[Fer01] J. Feret. Occurrence Counting Analysis for the Pi-Calculus. *Electronic Notes in Theoretical Computer Science*, 39(2):1–18, 2001.

[FMA13] A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS, pages 273–288. Springer, 2013.

[HO09] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[JHS12] 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 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

[MRR05] 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.

[PHW05] A. Di Pierro, C. Hankin, and H. Wiklicky. Quantitative static analysis of distributed systems. *Journal of Functional Programming*, 1:37–81, 2005.

[RL05] R.Gori and F. Levi. A new occurrence counting analysis for bioambients. In *APLAS*, volume 3780 of *LNCS*, pages 381–400. Springer, 2005.

[SBL11] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your Contexts Well: Understanding Object-Sensitivity. In *In Proc. of POPL'11*, pages 17–30. ACM, 2011.

[SPH10] J. Schäfer and A. Poetzsch-Heffter. JCobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, LNCS, pages 275–299. Springer, 2010.

[YBS86] A. Yonezawa, J.P. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. In *Procs. of OOPLSA'86*, pages 258–268, New York, USA, 1986. ACM.

A. Glossary of notation

Term	Section	Definition
$P(\overline{x}))$	2	A program P with input arguments $\overline{x}$
$executions(P(\overline{x}))$	2	The set of all possible fully expanded traces for $(P(\overline{x}))$
$o_{ij\ldots pq}$	2	$ij\ldots pq$ are allocation sites and $o_{ij\ldots pq}$ represents all possible runtime objects that were created at program point q when the enclosing instance method was invoked on an object represented by $o_{ij\ldots p}$
Operator $\oplus_k$	3	We use $ab\ldots c\oplus_k d$ for referring to the following object name: $ab\ldots cd$ if $\lvert ab\ldots cd\rvert \leq k$, or $b\ldots cd$ otherwise.
$pt(q, x)$	3	The set of object names at program point q for a given reference variable x obtained by the points-to analysis
o_l	3	An object with allocation sequence in l
o_λ	3	An object name obtained by the points-to analysis
$\mathcal{O}$	3	The set of all object names generated by the points-to analysis
$\mathcal{M}^I(b)$	3	The cost model for counting the number of instructions (see Definition 3.4)
$steps(t)$	3	The list of all steps executed by the trace t
$cost_P(t, o_l, \mathcal{M})$	3	The actual cost of a trace t of program P excecuted by object o_l with respect to the cost model $\mathcal{M}$
$UB_P^{\mathcal{M}}(\overline{x})$	3	The upper-bound expression obtained by applying the resource analysis for program $P(\overline{x})$ with respect to the cost model $\mathcal{M}$
$UB_P^{\mathcal{M}}(\overline{x})\rvert_{\mathcal{N}}$	3	The result of replacing in the upper-bound expression $c(o_\lambda)$ by 1 if $o_\lambda \in \mathcal{N}$ and by 0 otherwise
$root(o_l)$	4	Returns the root object of the cobox that owns o_l
$cobox\_roots(t)$	4	The multiset of cobox roots created during the execution of trace t
$obj\_in\_cobox(o_l, t)$	4	The multiset of objects that the cobox root o_l owns in a trace t
$Conf_P(\overline{x})$	4	The configuration of the program $P(\overline{x})$ (see Definition 4.2)
$inst(o_l, P, \overline{x})$	4	The maximum number of instances of the object o_l that can be created in the execution of all possible traces of program $P(\overline{x})$ (see Definition 4.4)
$Comm_P(\overline{x})$	4	The maximum number of method invocations that can be performed for all possible traces of program $P(\overline{x})$ (see Definition 4.6)
$ninter(o_l, o_{l'}, m, P, \overline{x})$	4	The maximum number of method invocations from object o_l to $o_{l'}$ by calling m (see Definition 4.8)
G_P	5	The points-to graph of the program P (see Definition 5.2)
$is\_root(o_\lambda)$	5	*is_root* decides whether the object name o_λ represents a cobox or not
$\mathcal{A}_P$	5	The abstract configuration of the program P (see Definition 5.4)
$cobox(o_\lambda, G_P)$	5	The set of all object names that belong to cobox o_λ in G_P
$covers(\mathcal{N}, O)$	5	$\mathcal{N}$ is a set of object names and O is a set of objects. Function *covers* returns true if all objects in O is covered by at least one object name in $\mathcal{N}$
$\mathcal{M}^C(b)$	5	$\mathcal{M}^C(b)$ is the cost model for counting the number of instances (see Definition 5.8)
I_P	5	The interactions graph of the program P (see Definition 5.12)

Term	Section	Definition
$\mathcal{M}^K(b)$	5	$\mathcal{M}^K(b)$ is the cost model for counting the number of communications performed(see Definition 5.13)
$\mathcal{D}_P(\overline{x})$	6	The distribution function (see Definition 6.4)
$\mathcal{K}_P(\overline{x})$	6	The communication function (see Definition 6.6)
$\mathcal{B}_P(\overline{x})$	6	The balance function (see Definition 6.8)
$\mathcal{T}_P(\overline{x})$	6	The target function