Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D3.3.2
Resource Analysis

Date of document: T24

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **UCM**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Resource Analysis

This document summarises the final version of deliverable D3.3.2 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

Deliverable D3.3.2 presents the main concepts and settings of the resource analyzer SACO after the first two years of the project. After a brief introduction with the objectives of this task, we present the basic concepts and describe the resource consumption properties which SACO is able to infer. We then provide a detailed user manual with the different options for running SACO within the Envisage Collaboratory.

List of Authors

Elvira Albert (UCM)
Jesús Correas (UCM)
Antonio Flores Montoya (TUD)
Enrique Martin-Martin (UCM)
Guillermo Román-Díez (UCM)

Contents

Chapter 1

Introduction

One of the most important features of a software system is its resource consumption. By resource, we mean not only traditional cost measures (e.g., number of executed instructions, or memory consumption) but also more advanced measures (e.g., number of tasks spawned, number of requests to remote servers, amount of data transmitted among the locations of a virtual system). Resource analysis (a.k.a. *cost* analysis) aims at *statically* inferring approximations of the resource consumption of executing the software system by just inspecting the code, in contrast to *dynamic* analysis in which the system is run for arbitrary inputs.

The *S*tatic *A*nalyzer for *C*oncurrent *O*bjects (SACO), presented in this deliverable, addresses objective O5 "Model Analysis Demonstrator" of the ENVISAGE project. This objective includes an automated analysis to check termination and obtain upper bounds on resource consumption including the amount of data transmitted between distributed components. Furthermore, SACO directly addresses objective O3.3 and is also relevant for O3.4, due to the fact that the invariant reasoning could benefit of this information, and thereby proves that the derived resources bounds are correct. The main challenges in those tasks that involve resource analysis are related to leveraging the system level and the deployment descriptions contained in the models to the resource analyzer, as we will explain in the next chapter. In particular, SACO is directly related to Task T3.3, whose goal consists in developing a systematic resource analysis for inferring an upper bound on the resource consumption of *abstract behavioral specification models* (ABS) at early stages of software development. With the information provided by SACO, we aim at having anticipated knowledge on the resource consumption of the different components which constitute the system. This can be crucial, among other things, to detect potential bottlenecks in the systems, or for optimally distributing the load of work. Besides, the information provided by the resource analyzer integrated in SACO can be used to compare the different deployment scenarios and drive the definition of the SLA of the application. In case of a SLA previously established, such information allows us to validate that the abstract system model complies with the SLA. Additionally, the results of the resource analysis could help to improve automation in the Verification Task T3.2.

The rest of the deliverable is structured in two chapters and several appendices that include the articles described in this document. Chapter 2 comprises a description of the different analyses that SACO integrates as follows:

- Section 2.1 introduces the starting point to this work, which is a rather limited resource analysis for ABS models which does not take system level and deployment descriptions into account.
- Section 2.2 presents the extensions required to handle concurrent executions in a precise way. Our first extension to the basic analysis is to use a *may-happen-in-parallel* pre-analysis in order to take system level information into account. We show in Section 2.2.1 how we can consider deployment decisions about the scheduling policy statically (this work has been published in LPAR-19 [15], included as Appendix A to this deliverable). Section 2.2.2 introduces another important extension to the basic framework to reason accurately about the resource consumption in the presence of interleaving tasks in a distributed system (published at ATVA'13 [13] and included as Appendix B). Section 2.2.3 presents an extension of the may-happen-in-parallel analysis that infers accurate information about the concurrent

behaviour in the system in the presence of inter-procedural synchronization, i.e., when future variables are passed as arguments of the tasks (published at SAS'15 [14] and included as Appendix C).

- Section 2.3 presents the extensions required to infer system level information about distributed systems in a precise way. Section 2.3.1 describes the performance indicators which allow us to obtain relevant information about distributed systems such us whether the load in the system is well-balanced (i.e., all distributed nodes execute a similar number of steps), or communication costs computed by overapproximating the sizes of the data transmitted between the locations of a distributed, or virtual system (the main results are published at [9] and ISOLA'14 [8], included as Appendices D and E to this deliverable). In Section 2.3.2 we introduce a novel resource analysis which is able to estimate the maximum of the resource consumption for non-cumulative resources that increase and decrease along the computation (i.e., resources can be acquired and released). This work has been published in TACAS'15 [11] and included as Appendix F. Section 2.3.3 presents the first static analysis to infer the peak cost in a virtual system, i.e., it infers the maximum amount of resources which each virtual location might have pending to execute along any execution of the system (this work has been published at SAS'14 [10] and included as Appendix G). In Section 2.3.4, we present a static analysis to infer the parallel cost of a distributed system, i.e., a new notion of cost that takes into account that some tasks execute in parallel across different distributed components and thus the total cost is the maximum among them. The main results related to parallel cost have been published in SAS'15 [7] and included as Appendix H to this deliverable. Section 2.3.5 describes a resource analysis of timed ABS models. That is, ABS models with time annotations that specify how time passes during the model's execution. The obtained results do not represent the total resource consumption of the analyzed system but rather the resource consumption of the system at a given time. This work is available at [19] and has been submitted for publication. It is included as Appendix I to this deliverable.

- Finally, in Section 2.4 we present CoFloCo, a new cost analysis backend that attempts to overcome some of the limitations of PUBS (the current backend [5]). CoFloCo can obtain precise upper bounds for many systems where PUBS fails. This work has been published in APLAS'14 [18] and is included as Appendix J to this deliverable.

After having introduced the theory underlying the resource analyzer, Chapter 3 provides a user manual in which the different settings and options are described in detail (the tool has been demonstrated at TACAS'14 [3] and a system description has been published in the conference proceedings, Appendix K).

Chapter 2

Basic Concepts

2.1 Starting Point

Resource analysis (or cost analysis [6]) aims at statically, i.e. without running the program, bounding the cost of executing programs for any possible input data value. Such bounds are given as functions on the input data sizes. This section briefly describes the starting point for our work: a basic resource analysis for ABS models [4] which is able to separate the cost of the distributed components of the system using *cost centers* and which discards information on the *shared data* at processor release points. In the following we introduce the basic concepts of resource analysis illustrating them by means of examples.

The first challenge for computing the cost of a program is to have an automatic method to compute the maximum number of iterations on loops (or recursive structures), and, as a side effect, to guarantee the termination of the program. The use of ranking functions to automatically bound the number of iterations was proposed in [5]. A ranking function is a function on the program variables which (1) is positive and (2) decreases in each iteration, and thus, can be used to bound the number of loop iterations. Once we have bounded the number of iterations, in order to determine the cost of executing the program, we define the notion of resource that we aim to approximate by static analysis. Typical examples of resources that can be measured include: number of execution steps, amount of memory allocated, amount of data transmitted over the Internet, etc. In order to define a generic resource analyzer we rely on the notion of *cost model*, which determines the type of resource to be measured. A cost model assigns a cost to the different instructions of the programming language. For instance, if we want to count the number of executed steps, all instructions are assigned cost 1; if we want to measure the amount of memory allocated, only the instructions which allocate memory are assigned a cost, which is usually determined by the size of the created data; if we want to count the number of method invocations, only the instructions which invoke methods are assigned cost 1, all remaining instructions have a cost of 0. Let us see an example of resource analysis for the program in Figure 2.1.

Example 1 *Figure 2.1 shows the ABS source code of the example that will be used in some parts of the rest of this chapter. It includes a class* PrettyPrinter*, which displays some information, and a class* VendingMachine*, with methods to insert a number of coins and to retrieve all coins on a vending machine. The program includes methods* showIncome *and* showCoin*. We assume that both methods are not empty and they do not contain any method call. Let us start by studying the resource consumption of the loop at line 23 (for brevity, denoted L23) in method* retrieveCoins*. We need to ignore at this point the* await *instruction at L26, otherwise the basic framework cannot find an upper bound. Ignoring* await*, it is clear that the loop iterates* coins *times. Hence, the local ranking function* $\mathsf{nat}(\mathsf{coins})$[1] *is computed, where* $\mathsf{nat}(x) = x$ *if* $x \geq 0$*, and* $\mathsf{nat}(x) = 0$ *if* $x < 0$*. We then need to find an over-approximation of the cost of executing any iteration of the loop at L23, denoted* C_l*. Note that, as we are computing the cost executed within* retrieveCoins*,* C_l *does not include the cost executed by* showCoin*. Thus, the cost of executing the loop is bounded by the expression* $\mathsf{nat}(\mathsf{coins}) * C_l$*.*

[1] We use the nat function to avoid negative values in the ranking function, since coins is an integer variable.

```
1  class PrettyPrinter{
2    Unit showIncome(Int n){...}
3    Unit showCoin(){...}
4  }//end class
5
6  class VendingMachine{
7  Int coins;
8  PrettyPrinter out;
9    Unit insertCoins(Int n){
10     Fut⟨Unit⟩ f;
11     while (n>0){
12       n=n−1;
13       f=this ! insertCoin();
14       await f?;
15     }
16   }
17   Unit insertCoin(){
18     coins=coins+1;
19   }
20   Int retrieveCoins(){
21     Fut⟨Unit⟩ f;
22     Int total=0;
23     while (coins>0){
24       coins=coins−1;
25       f=out ! showCoin();
26       await f?;
27       total=total+1;
28     }
29     return total;
30   }
31 }//end class
32 //main method
33 main(Int n){
34   PrettyPrinter p;
35   VendingMachine v;
36   Fut⟨Int⟩ f;
37   p=new PrettyPrinter();
38   v=new VendingMachine(0,p);
39   v ! insertCoins(n);
40   f=v ! retrieveCoins();
41   await f?;
42   Int total=f.get;
43   p!showIncome(total);
44 }
```

Figure 2.1: Example of an ABS Program

An upper bound of the cost of method retrieveCoins *can be easily obtained by adding the cost of the loop plus that of the instructions outside the loop (C_o), that is, $C_{\mathsf{retrieveCoins}} = C_o + \mathsf{nat}(\mathsf{coins}) * C_l$. Let us observe that this cost is not constant – it depends on the maximal value of field* coins*. Expressions C_o and C_l are constant, but their values depend on the resource of interest. For instance, if we use the cost model that counts executed instructions, then $C_o = 3$ (instructions L21, L22 and L29), and $C_l = 4$ (instructions L23–L28). An upper-bound on the number of executed instructions by* retrieveCoins *is $3 + \mathsf{nat}(\mathsf{coins}) * 4$. —recall that the cost executed at* showCoin *is not included in C_l. Other cost models can be applied as well. As an example, we can count the number of tasks spawned in* retrieveCoins *by means of a cost model that assigns 1 to L25 and 0 to the rest of the instructions. Then, the expression $0 + \mathsf{nat}(\mathsf{coins}) * 1$ bounds the number of calls performed by* retrieveCoins*— remember that* showCoin *does not contain any method call. Finally, let us note that the cost model to measure the amount of memory allocated by* retrieveCoins *would assign cost 0 to all instructions in the method, since no object is created at* retrieveCoins*. Similarly, we can obtain $C_{\mathsf{insertCoins}}$, $C_{\mathsf{insertCoin}}$,.., for all methods.*

In a distributed setting with multiple objects possibly running concurrently on different central processing units (CPUs), the notion of cost has to be extended so that, rather than aggregating together the cost of all execution steps, they are split among the different computing infrastructures. This is important since different infrastructures might have different configurations, such as CPU, memory, etc. Therefore, inferring the resource consumption for each infrastructure separately helps in identifying those that might exceed the resource limit (e.g., run out of memory). With this aim, we adopt the notion of cost centers proposed in [4]. A cost center is a symbolic artifact of the form $c(o)$ that we include in the cost expressions to distribute the cost associated to the object o. We will use as object identifier the program point where the object is created. Since the concurrency unit of our language is the object, cost centers are used to assign the cost of each execution step to the cost center associated to the object where the step is performed.

Example 2 *Let us study the resource consumption of the* main *method in Figure 2.1. Three objects are involved in the execution: the object that executes the* main*, and the two objects created resp. at L37 and L38. The associated cost-centers are denoted respectively $c(\epsilon)$, $c(37)$ and $c(38)$. We use C_{m} to represent the upper bound of the cost executed at method* m*, including the number of times that* m *is executed. For instance, if we assume that one call to* showCoin *executes 5 instructions, as* showCoin *is executed* coins *times, we have that $C_{\mathsf{showCoin}} = \mathsf{nat}(\mathsf{coins}) * 5$. The following expression bounds the cost of the* main *method at the level of distributed components (i.e. objects):*

$$c(\epsilon) * C_{\mathsf{main}} + c(38) * C_{\mathsf{insertCoins}} + c(38) * C_{\mathsf{insertCoin}} + c(38) * C_{\mathsf{retrieveCoins}} + c(37) * C_{\mathsf{showCoin}} + c(37) * C_{\mathsf{showIncome}}$$

This allows obtaining the cost attributed to a concrete object o by replacing the associated cost center $c(o)$ by 1, and replacing by 0 the remaining cost centers. E.g., the cost attributed to the object created at L37 is bounded by the expression $c(37) * C_{\mathsf{showCoin}} + c(37) * C_{\mathsf{showIncome}}$.

One of the main challenges when (statically) analyzing concurrent programs is to model the behavior of the shared memory. Let us consider the concurrent execution of two tasks A and B on an object o. While A is executing, task B can interleave its execution with it and modify the values stored in o's heap, affecting A's behavior and possibly its resource consumption. Therefore, A cannot assume that the values stored in o's heap are preserved. A safe solution to this problem [4] consists in discarding all information about fields at all release points. As a consequence of this loss of information, cost analysis can be rather imprecise in many situations or even cannot succeed in finding the ranking function for some loops (therefore not being able to compute an upper bound).

Example 3 *Let us study again the cost of the loop at L23, but this time without ignoring the* await *at L26. The important point to note is that, due to the* await *at L26, the processor might be released allowing other tasks to interleave their execution with the execution of* retrieveCoins*, possibly changing the value of the shared variable* coins*. The safe solution consists in discarding all information about* coins *at this point, i.e., after executing the* await *at L26 the shared variable* coins *can have any value. This means* nat(coins) *is not a valid ranking function for the loop. Consequently, without further improvements, the termination of such loop cannot be guaranteed and its cost cannot be bounded with the available techniques.*

2.2 Handling Concurrency

2.2.1 Use of May-Happen-in-Parallel (with Priorities)

As mentioned in the previous section, if a loop contains an await instruction, then another task can take the processor in the middle of the execution and change the shared memory. If we do not have any information about the concurrent behavior of the program, we have to assume that every shared variable may change at this await point. In order to handle situations like these, we will use the information provided by an analysis called *may-happen-in-parallel* (MHP). MHP is an analysis which has been defined for many concurrent languages (see [1, 16, 22, 23]). We use the MHP analysis defined for the ABS language in [12]. This analysis takes into account the system level information of the application learned from the task invocations and release instructions, and returns a list of all pairs of program points that can be executed in parallel or that can interleave their execution. This information is crucial in this context since it allows us to know whether or not the value of a specific shared variable may be modified when the execution of a loop releases the processor at an await instruction. If the shared memory cannot be modified, then we can confirm that the ranking function obtained ignoring the await instruction is indeed a correct ranking function considering concurrent interleavings. Otherwise, as the shared memory could be modified, the mentioned ranking function may be incorrect and the analysis would fail as in the scenario presented in Section 2.1.

Example 4 *The MHP analysis applied over the program in Figure 2.1 infers (among other information) that the instructions at L10–L16 and L18 can interleave with the execution of the loop when the processor is released at L26 (*await *instruction). Since the instruction at L18 modifies the shared variable* coins*,* nat(coins) *will not be a valid ranking function for the loop in* retrieveCoins*. In this case, the information from the MHP analysis does not improve the results and the resource analysis fails, as the original analysis.*

Let us observe that any improvement in the precision of the results of the MHP analysis could improve the precision of the results of the resource analysis as well (await instructions could interleave with a smaller set of instructions). For instance, the precision of the MHP analysis can significantly improve by taking advantage of deployment decisions about the scheduling policy of the system. We have studied the extension of the MHP analysis with priority-based scheduling, one of the policies more commonly adopted in practice. In this policy, every task invocation is assigned a priority number, which can be seen as an additional argument

in the call. When the processor is idle, the scheduler selects and executes the task with the highest priority in the current object, instead of taking *non-deterministically* some pending task.

Example 5 *Consider the program in Figure 2.1 with the following additional priorities to task invocations: the task invocation at L40 (*retrieveCoins*) has the maximum priority* 10*, and the task invocations at L39, L13 and L25 (*insertCoins*, *insertCoin* and *showCoin* respectively) have the minimum priority* 0*. Using this priority information the MHP analysis can obtain more precise results: when the execution of the loop releases the processor at L26, task* insertCoin *cannot interleave with it, because it has a lower priority. As the* await *instruction at L26 cannot interleave with any other instruction that modifies the shared memory, its value will not change, and thus the ranking function* nat(coins) *is correct for the loop in* retrieveCoins*. Hence we can bound its resource consumption.*

Although the extensions presented in this section have been implemented and can improve over the original resource analysis presented in the Section 2.1, they are currently not available in the SACO tool —see Section 3. The reason is twofold: (1) currently the ABS language does not support setting the priority of task invocations, and (2) this extension is further improved by the extended termination and resource analysis presented in the next section, which is fully implemented and integrated within the tool.

2.2.2 Rely-Guarantee Resource Analysis

As we have seen in Example 4, the interleaving information computed by the MHP analysis is not always enough to obtain ranking functions for loops. The previous section showed that if the MHP analysis can ensure the absence of interleavings that update the shared memory, the ranking function obtained by ignoring the await is still valid. In this section we will show that using the MHP information in a more sophisticated way we can assure the termination and bound the resource consumption of loops even in some cases when interleavings modify the shared memory. We will focus first on termination, and then we will extend the results to resource consumption.

The main idea for proving termination of loops with concurrent interleavings is the following: if (1) a loop is terminating *assuming* there are no interleavings and (2) we can prove that the possible interleavings modify the shared memory a *finite number of times*, then the loop is terminating even with concurrent interleavings. This can be seen clearly in the loop of retrieveCoins in Figure 2.1.

Example 6 *According to the MHP analysis, the* await *instruction at L26 can happen in parallel with the instruction at L18, which modifies the shared variable* coins*. When the updating instruction interleaves with the loop, the shared variable may decrease or even increase between iterations, potentially causing divergence. However, if we can assure that the updating instruction is executed a finite number of times, then, from some point in the execution on, the potential interleavings will not change the shared memory any longer. Therefore, from that point on the shared variable* coins *will decrease at each iteration and the loop will terminate.*

It is interesting to note that checking if an instruction is executed a finite number of times is easily reduced to checking if the loop that contains that instruction is terminating, so we have a recursive procedure for termination. This approach is inspired by the *rely-guarantee* style of reasoning used for compositional verification and analysis of thread-based concurrent programs: we first assume a property on the global state in order to prove termination of a loop, and then, we prove that this property holds. Concretely the termination of a loop L involves:

1. Check the termination of L *assuming* that the shared memory is modified a finite number of times.
2. For every instruction that modifies the shared memory and can interleave with L, check the termination of the loop that contains it. This second step proves the property assumed in step 1.

We can prove that a program is terminating by applying this approach to every loop. Terminating programs have the mentioned property that the size of all data is bounded. Concretely every shared variable f will be bounded by some lower and upper bounds: $\mathsf{f} \in [\mathsf{f}^{-}..\mathsf{f}^{+}]$. The existence of these bounds is important in other analysis like resource consumption, as we will see.

Example 7 *Consider again the loop in method* retrieveCoins *in Figure 2.1. As mentioned in Example 6, this loop is terminating if the instruction at L18 is executed a finite number of times. Following the rely-guarantee reasoning, we need to prove this property. The instruction at L18 is executed by method* insertCoin*, which is invoked in the loop of method* insertCoins *at L13. Therefore we have to prove the termination of this second loop (L11–L15). This case is simpler since it depends on a local variable* n *which cannot be modified due to possible interleavings. The loop will therefore be executed exactly* $\mathsf{nat}(n)$ *times. Consequently, the instruction at L18 will modify the shared variable* coins *a finite number of times, and the loop in* retrieveCoins *will be proved terminating.*

We can use similar reasoning to measure the resource consumption of programs. In this case we want to compute *precise* ranking functions to bound the number of iterations of loops *considering the possible interleavings*. Once we have these ranking functions we can compose them with the cost models and cost centers presented in Section 2.1 in order to obtain the desired resource consumption bounds. To obtain such ranking functions we combine the ranking functions ignoring the interleavings (obtained by the original analysis presented in Section 2.1) with the number of times that the shared memory is modified.

Example 8 *Consider again the loop in* retrieveCoins *in Figure 2.1. As we have mentioned in Example 1, if we ignore the interleavings, the ranking function is* $\mathsf{nat}(\mathsf{coins})$*. However, each time there is an interleaving the value of the shared variable* coins *might change. The only interleaving instruction that modifies* coins *is in L18, which is executed* n *times. In order to obtain an upper bound, we will take the worst case: the loop in* retrieveCoins *will execute all its iterations, but then in the last iteration an interleaving will replace the current value of* coins *by its maximum value* coins^+*. This situation will happen exactly* n *times, because the updating instruction at L18 is visited* n *times, so the final ranking function for the loop in* retrieveCoins *will be* $\mathsf{nat}(\mathsf{coins}^+) * \mathsf{nat}(n)$*—note that the upper bound* coins^+ *exists because the program has been proved terminating.*
With this ranking function we can obtain an upper bound for retrieveCoins *using any desired cost model. For example, if we want to count executed instructions, the upper bound will be* $3 + \mathsf{nat}(\mathsf{coins}^+) * \mathsf{nat}(n) * 4$*, since there are* 3 *instructions before the loop and* 4 *instructions inside it—recall that the method* showCoin *is empty. If, on the other hand, we use a cost model to count spawned tasks, the upper bound will be* $0 + \mathsf{nat}(\mathsf{coins}^+) * \mathsf{nat}(n) * 1$ *because there are* 0 *task invocations before the loop and exactly one invocation (L13) inside it. Similarly, we could add cost centers in order to associate resource consumption to the object where the step is performed.*

The extensions for termination and resource consumption presented in this section have been both integrated within the SACO tool delivered with this document. Section 3 contains a detailed explanation of their usage and how to visualize their results.

2.2.3 May-Happen-in-Parallel with Inter-Procedural Synchronization

The ABS language allows passing future variables as arguments of tasks. This is an important feature as it enables inter-procedural synchronization, i.e., synchronizing with the termination of a task outside the scope in which the task is spawned. The original MHP analysis of [12] only handles a restricted form of intra-procedural synchronization and we have worked on the extension of this analysis to inter-procedural synchronization.

Example 9 *If the MHP analysis of [12] is applied to the program in Figure 2.2, it will infer that program point at L7 can execute in parallel with L12 or L8 can execute in parallel with L19. However, it is not able to infer that L9 and program point at L12 cannot happen in parallel. To infer this, we need to track inter-procedural synchronizations.*

When analyzing **await** y? in the program of Figure 2.2 using the previous MHP, the analysis marks the task bound to y as finished, but it does not track the inter-procedural information, i.e, it does not mark the

```
1  Unit main(){
2    Fut<Unit> x;
3    Fut<Unit> y;
4    o1 = new O1();
5    o2 = new O2();
6    x = o1!f();
7    y = o2!g(x);
8    await y?;
9  }

10 class O1{
11   Unit f(){
12     skip;
13   }
14 }
15
16 class O2{
17   Unit g(Fut<Unit> w){
18     skip;
19     await w?;
20     skip;
21   }
22 }
```

Figure 2.2: Example of Inter-Procedural Synchronization

task bound to x as finished. In order to overcome this imprecision, we have developed a must-have-finished analysis that captures inter-procedural synchronization. This analysis returns for each program point a set of future variables that must have finished when reaching this program point. All technical details about this analysis have been published in the proceedings of SAS'15 [14].

Example 10 *The must-have-finished analysis applied to the program in Figure 2.2 will infer that when reaching L19, it is guaranteed that whatever task bound to* w *has already finished, when reaching L7 there is no task finished and when reaching L8, it is guaranteed that whatever tasks bound to* x *and* y *have already finished.*

This information can be integrated in the MHP analysis improving its precision, namely when an **await** x? statement is reached, it marks as finished not only the task bound to x but also the task whose termination depends on the task bound to this future variable. Observe that we sometimes omit variable types in examples for readability.

Example 11 *If the MHP analysis with inter-procedural synchronization is applied to the program in Figure 2.2, it will infer that L7 can happen in parallel with L12, or L8 can happen in parallel with L19, as the MHP analysis of Albert et al. [12] does. In addition, it is able to infer that program L9 cannot happen in parallel with L12 because the task bound to* y *has finished its execution and transitively the task bound to* x *too. It will also infer that L12 and L20 cannot happen in parallel since the execution of the task passed as parameter to method* ***g*** *is finished at L20.*

This extension is fundamental as MHP is an analysis of utmost importance to ensure both liveness and safety properties of ABS programs, namely the enhancement of the MHP analysis with inter-procedural synchronization allows improving the precision of the deadlock, termination and resource analysis.

The MHP analysis with Inter-Procedural Synchronization presented in this section is integrated within the SACO tool delivered with this document. Section 3 contains a detailed explanation of its usage in SACO and how to visualize its results.

2.3 Handling Distribution

2.3.1 Performance Indicators

In this section we define indicators that can be considered to estimate the performance of a distributed system [9]. In particular, we are interested in predicting the load balancing of the distributed objects, the number of communications between nodes and the amount of data transferred among them.

```
1  void m (int n) {
2   I a = new Obj();
3     while (n > 0) {
4       a!p(n,a);
5       n = n − 1;
6     }
7  }
```

```
8  void p (int n, I x) {
9     while (n > 0) {
10      x!q();
11      n = n − 1;
12    }
13 }
14 q () { 10 instr }
```

```
15 void m2 (int n) {
16    while (n > 0) {
17      I a = new Obj();
18      a!p(n,a);
19      n = n − 1;
20    }
21 }
```

Figure 2.3: Example of Performance Indicators

Load Balance

Using the cost centers described in Section 2.1, we define an indicator to assess how balanced the load of the distributed nodes that compose the system is. By attributing the cost of each instruction to the object responsible of executing it, upper bounds can help during the development process to take better design decisions for obtaining an optimal load balancing.

Example 12 *In the source code shown in Figure 2.3 (left and center), method* m *creates a new object at L2 by means of* new Obj(), *which is referenced by variable* a, *and the* while *loop spawns* n *tasks executing method* p *(L4). Besides, method* p *contains another loop that calls* q n *times (L10). Observe that the second argument of the call to* p *at L4 causes method* q *to be executed at object* a. *If we replace the second argument by* this *at L4, that is* a!p(n,this), *method* q *will be executed at the object executing* m. *We refer to this object as* ϵ. *The upper bound expressions for the number of steps are the same for both cases, but such decisions are crucial for properly balancing the system. By using the resource analysis of Section 2.1, for* a!p(n,a) *at L4, the cost attributed to* $c(\epsilon)$ *is* $9 + 7 * n$ *and the cost attributed to* $c(2)$ *is* $1 + n * (6 + 14 * n)$. *It can be seen that the program is not properly balanced, since the cost attributed to* $c(\epsilon)$ *is a linear expression w.r.t. the value of* n, *while the cost attributed to* $c(2)$ *is a quadratic expression w.r.t.* n. *On the other hand, by using* a!p(n,this) *at L4, we have that the cost attributed to* $c(2)$ *is* $1 + n * (6 + 7 * n)$ *and the cost attributed to* $c(\epsilon)$ *is* $9 + n * (7 + 7 * n)$. *In this case we can see that the program is more evenly balanced, as both expressions are quadratic w.r.t.* n.

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. As we have seen in the previous example, configurations may be straightforward in simple applications, but the tendency is to have rather complex and dynamically changing configurations (cloud computing is an example of this). To this end, in addition to the upper bound on the number of instructions executed by each object, it is required to have information about how many instances of each object might exist. Resource analysis described at Section 2.1 can also be extended to provide such information.

Example 13 *As we have seen in Example 12, method* m *only uses two objects,* ϵ *and* a. *In contrast, method* m2 *shown in Figure 2.3 (right) creates objects within a loop and, by means of the resource analysis, we can infer that the number of objects created at* m2 *is bounded by the value of* n.

If we consider a system to be optimally balanced when all its components execute the same number of instructions, we can use the upper bounds on the number of instructions and the upper bounds on the number of distributed components to reason about how balanced the load of the distributed nodes that compose the system is. As regards the number of instructions executed by each object in the system, we have to take into account that an abstract object might represent multiple concrete objects. This means that the number of instructions executed by an abstract object actually accounts for the instructions executed by all objects it represents.

Example 14 *The analysis of* m2 *returns that the cost attributed to* $c(\epsilon)$ *is* $6 + 10 * n$ *and that the number of instructions executed by the objects created within the loop is* $n * (7 + 14 * n)$. *As we have seen in Example 13,*

the number of objects created within the loop is bounded by the value of the input argument n. Therefore, we have n objects, identified by the cost center c(17) that execute $n*(7+14*n)$ instructions, and another object, identified by the cost center $c(\epsilon)$, that executes $6 + 10 * n$ steps, which implies that the system is properly balanced as all objects perform a linear number of instructions.

Section 3 contains a detailed explanation of how to use the performance indicators implemented in SACO and how to visualize its results.

Transmission data sizes

Another relevant contribution for Envisage is the extension of SACO to infer the transmitted data sizes. We have developed a static analysis to infer the amount of data that a distributed system may need to transmit. Knowledge of the transmission data sizes is essential, among other things, to predict the bandwidth required to achieve a certain response time, or conversely, to estimate the response time for a given bandwidth. The different locations of a distributed system communicate and coordinate their actions by posting tasks among them. A task is posted by building a message with the task name and the data on which such task has to be executed. When the task completes, the result can be retrieved by means of another message from which the result of the computation can be obtained. Thus, the transmission data size of a distributed system mainly depends on the amount of messages posted among the objects of the system, and the sizes of the data transferred in the messages. In order to estimate the transmission data sizes, we need to keep track of the amount of data transmitted in two ways:

1. By posting asynchronous tasks among the objects. This requires building a message in which the name of the task to execute and the data on which it executes are included.

2. By retrieving the results of executing the tasks. In our setting, future variables are used to synchronize with the completion of a task and retrieve the result.

Our analysis infers a safe over-approximation of the transmission data sizes required by both sources of communications in a distributed system. Our method infers two different pieces of information: the number of tasks spawned at a given object, and the data sizes transmitted as a result of the task spawned.

Since we are considering an abstract representation of data by means of functional types, we will focus on *units of data* transmitted instead of bits, which depends on the actual implementation and is highly platform-dependent. Concretely, we assume that the cost of transmitting a basic value or a data type constructor is one unit of data. This size measure is known as *term size*. However, our static analysis would work also with any other mapping from data types to corresponding sizes (given by means of a function α).

Example 15 *The program shown in Figure 2.4 creates objects s and m at L56 and L57, respectively, to perform some processing on a list. The list l has an initial content set at L55 (not relevant for the example) that is passed as a parameter of the call to method work at object m, and thus there is data transmission at this point. Method work extends the list with n values, and calls method process at object s (L73) after adding each element to the list, passing the list as argument. Method process does some processing to the list passed as argument. There are two program points in method work where data is transmitted between objects m and s: L73 and L74, that correspond to the call to process and the retrieval of the returned value, respectively. Data structures are defined in ABS by means of data constructs, as shown in L51 with the data type definition for representing lists of integers. We consider the term size of data structures as the size measure. For example, a list defined as l = Cons(1, Cons(2, Cons(3,Nil))) has size $\alpha(l) = 7$, as it counts 2 for each element in the list (the Cons constructor and the element itself), plus 1 for the Nil constructor.*

For inferring an upper bound on the number of tasks spawned between all pairs of distributed objects, we use the cost analysis framework described in Section 2.1. In particular, we need to use a *symbolic* cost model which allows us to annotate the caller and callee objects when a task is spawned in the program. In essence, if we find an instruction a!m(x) which spawns a task m at object a, the cost model symbolically

```
51 data List = Nil | Cons(Int, List);
52 // main method
53 Unit main (Int n) {
54  Slave s; Master m; List l;
55  l = ...;
56  s = new Slave();
57  m = new Master(s);
58  m!work(l,n);
59 }
60 class Slave {
61  Int process (List le) {
62   ...
63   return h;
64 }}// end class
```

```
65 class Master {
66  Slave s;
67  work(List l,Int n) {
68   Int x;
69   Int n;
70   fut<Int> y;
71   while (n>0){
72    l = Cons(n,l);
73    y = s!process(l);
74    x = y.get;
75    n−−;
76   }
77  }
78 }// end class
```

Figure 2.4: Example of transmission data sizes

counts $c(this, a, m) * 1$, i.e., it counts that 1 task executing m is spawned from the current object this at a. If the task is spawned within a loop that performs n iterations, the analysis will infer $c(this, a, m) * n$.

Example 16 *For the code in Figure 2.4, cost analysis infers that the number of iterations of the loop in work (at L71) is bounded by the expression* $\mathsf{nat}(n)$*. Then, by applying the number of tasks cost model we obtain the following expression that bounds the number of tasks spawned at L73:* $c(\mathsf{m}, \mathsf{s}, \mathsf{process}) * \mathsf{nat}(n)$

The second piece of information obtained by our analysis is the data size transmitted as a result of spawning a task. To this end, we need to infer the sizes of the arguments in the task invocations. Typically, size analysis [17] infers upper bounds on the data sizes at the end of the program execution. Here, we are interested in inferring the sizes at the points in which tasks are spawned. In particular, given an instruction a!m(x), we aim at over-approximating the size of x when the program reaches the above instruction. If the above instruction can be executed several times, we aim at inferring the largest size of x, denoted $\alpha(x)$, in all executions of the instruction. Altogether, $c(this, a, m) * \alpha(x)$ is a safe over-approximation of the data size transmission contributed by this instruction. The analysis will infer such information for each pair of objects in the system that communicate, annotating also the task that was spawned.

Example 17 *Since in method work the size of l is increased within the loop at L72, the maximum size of l is produced in the last call to process. Recall that the term size of the list l counts 2 units for each element in the list. Therefore, each iteration of the loop at L71 increments the term size of the list in 2 units and, consequently, the last call to process is executed with a list of size* $l_0 + 2 * n$*, where* l_0 *is the term size of the initial list, created at L55. In addition, the value returned by the call to process is retrieved at L74. Since the data retrieved is of type Int, its size is 1.*
Then, the data transmitted between objects m and s is bounded by the following expression, where the constant $\mathcal{I}$ *is the size of establishing the communication:*

$$c(\mathsf{m}, \mathsf{s}, \mathsf{process}) * \mathsf{nat}(n) * (\mathcal{I} + \mathsf{nat}(l + n * 2)) + c(\mathsf{s}, \mathsf{m}, \mathsf{process}) * \mathsf{nat}(n) * (\mathcal{I} + 1)$$

The analysis for inferring transmission data sizes is integrated within the SACO tool delivered with this document. Section 3 contains a detailed explanation of its usage and how to visualize its results.

2.3.2 Non-cumulative Cost

Another extension to sequential resource analysis is the inference of non-cumulative resources [11]. Existing cost analysis frameworks have been defined for cumulative resources which keep on increasing along the execution. In contrast, non-cumulative resources are acquired and (possibly) released along the execution.

```
1 Unit main (Int s, Int n) {
2   x = acquire(k1);
3   r = acquire(k2);
4   r = acquire(s);
5   release r;
6   y = acquire(n);
7   release x;
8 }
```

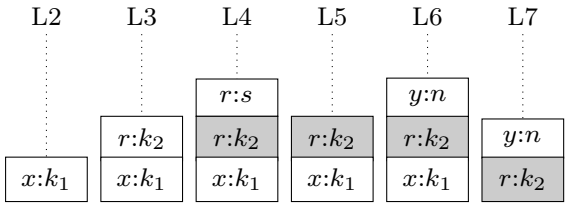


Figure 2.5: Example of an ABS Program with Non-Cumulative Resources

Examples of non-cumulative cost are memory usage in the presence of garbage collection, number of connections established that are later closed, or resources requested to a virtual host which are released after using them.

To model non-cumulative resources we have added two instructions to the language: r = acquire(e), which acquires the amount e of resources, referenced by the variable r, and the instruction release r, to free the resources pointed by r.

It is recognized that non-cumulative resources introduce new challenges in resource analysis [2, 20]. This is because the resource consumption can increase and decrease along the computation, and it is not enough to reason on the final state of the execution, but rather the upper bound on the cost can happen at any intermediate step. The analysis of non-cumulative resources is defined in two steps: (1) We first infer the sets of resources which can be in use simultaneously (i.e., they have been both acquired and none of them released at some point of the execution). This process is formalized as a static analysis that (over-)approximates the sets of acquire instructions that can be in use simultaneously, allowing us to capture the simultaneous use of resources in the execution. (2) We then perform a *program-point* resource analysis which infers an upper bound on the cost at the points of interest, namely the points at which the resources are acquired. This is done by using as cost centers the program points at which the resources are acquired so that they are separated within the upper bound from the other acquired resources. From such upper bounds, we can obtain the *maximum* cost by just eliminating the cost due to acquire instructions that do not happen simultaneously with the others (according to the analysis information gathered at step 1). This analysis can be extended to multiple resources by adding a parameter to acquire instructions which determines the kind of resource to be allocated, i.e, acquire(res,e) where res specifies the type of resource.

Example 18 *Figure 2.5 shows a sample program where some non-cumulative resources are acquired at L2, L3, L4 and L6, and released at L5 and L7. At L4 variable* r *is assigned without previously releasing the resources acquired at L3, producing a resource leak (shown in grey). At L5 the program releases the resources acquired at L4. At L6* n *resources are acquired, and since they are not released, these resources leak upon program termination. Finally, at L7 the resources acquired at L2 are released. To the right of Figure 2.5 we show how resources are actually acquired and released during execution. We can see that the program points where more resources are simultaneously acquired are L4 and L6. Our analysis infers for each program point pp the set of program points corresponding to acquire resources that have not been released in any execution reaching pp. The analysis is similar to a liveness analysis since release instructions allow detecting that a resource is freed. The program points that correspond to maximal sets lead to the maximum resource consumption. In this example, they are:* $A_1 = \{L2, L3, L4\}$ *and* $A_2 = \{L2, L3, L6\}$*, that correspond to the resources alive at lines L4 and L6 of the execution, respectively. By means of the program-point resource analysis, we can infer that the amount of resources acquired by the sets* A_1 *and* A_2 *are bounded by the expressions* $N_{A_1} = k_1 + k_2 + \mathsf{nat}(s)$ *and* $N_{A_2} = k_1 + k_2 + \mathsf{nat}(n)$*. Therefore, the non-cumulative resource consumption of the program is bounded by the expression* $max(N_{A_1}, N_{A_2})$*.*

Section 3 contains a detailed explanation of its usage in SACO and how to visualize its results.

2.3.3 Peak Cost

One of the important results obtained in the area of resource analysis during the first two years of the project is the extension of the SACO system to infer the *peak cost*. Existing cost analyses for distributed systems infer the *total* resource consumption [4] of each distributed object, e.g., the total number of instructions that it needs to execute, the total amount of memory that it will need to allocate, or the total number of tasks that will be added to its queue. This is a too pessimistic estimation of the amount of resources actually required in the real execution. An important observation is that the *peak* cost will depend on whether the tasks that the object has to execute are pending *simultaneously*. We aim at inferring such peak of the resource consumption which captures the maximum amount of resources that the object might require along any execution. This information is crucial to dimensioning the distributed system: it will allow us to determine the size of each object's *task queue*; the required size of the object's memory; and the processor execution speed required to execute the peak of instructions and provide a certain response time. It is also of great relevance in the context of virtualization as used in cloud computing, as the peak cost allows estimating how much processing/storage capacity one needs to buy in the host machine, and thus can help reduce costs.

We have developed a static analysis to infer the peak of the resource consumption of ABS models, which takes into account the type and amount of tasks that the distributed objects can have in their queues simultaneously along any execution, to infer precise bounds on the peak cost. The first step of the peak cost analysis is the *total cost analysis* described in Section 2.1. Then, an *abstract queue configuration* is inferred for each object, which captures all possible configurations that its queue can take along the execution. A particular queue configuration is given as the sets of tasks that the object may have pending to execute at a moment of time. We rely on the information gathered by the MHP analysis mentioned in Section 2.2.1 to define the abstract queue configurations.

Example 19 *The upper bound expression inferred by the resource analysis in Example 2 for the cost center* $c(37)$ *is* $c(37) * C_{\mathsf{showCoin}} + c(37) * C_{\mathsf{showIncome}}$*, where* C_{showCoin} *and* $C_{\mathsf{showIncome}}$ *represent the computed upper bound of the cost of executing all instances of the respective tasks, and it is the total cost associated to the object created at L37. This estimation is pessimistic w.r.t. the actual resources required for object L37. The calls to* showCoin *at L25 are awaited at L26, and therefore when* retrieveCoins *finishes its execution, all tasks spawned at L25 have finished as well. This means that when* showIncome *is called at L43, the queue of the object L37 is guaranteed to be empty. Therefore, object L37 only requires resources to either execute* showIncome *or execute (several instances of)* showCoin*.*
In contrast, the object created at L38 will execute asynchronous calls to insertCoins *and* retrieveCoins *that may be posted to the object queue simultaneously. Consequently, the object must have enough resources for dealing with both calls at the same time.*

A *quantified queue configuration* is inferred for each object, which over-approximates the peak cost of each distributed object. For a given queue configuration, its quantified configuration is computed by removing from the total cost expression those tasks that do not belong to that configuration. The peak for the object is the maximum of the costs of all configurations that its queue can have.

Example 20 *Since object L37 cannot have in its queue calls to* showCoin *and to* showIncome *simultaneously, an upper bound of the peak cost of this object is* $max(C_{\mathsf{showCoin}}, C_{\mathsf{showIncome}})$*, where* C_m *bounds the cost executed by method* m*. Observe that since* showCoin *is invoked several times inside the while loop at L25,* C_{showCoin} *bounds the execution of all such instances.*

The accuracy of the peak analysis can be improved if it is possible to determine that, for a method that is called several times, there will be no more than one task in the queue for that method at any time. The MHP analysis can detect whether several instances of a method m are never queued simultaneously. In that case, only one instance of the method can be considered for the peak cost. Under some specific technical conditions, it is sound to obtain the cost of executing one instance of m, by dividing the bound C_m obtained for the total cost executed by m, by the number of instances spawned for m. The number of instances can

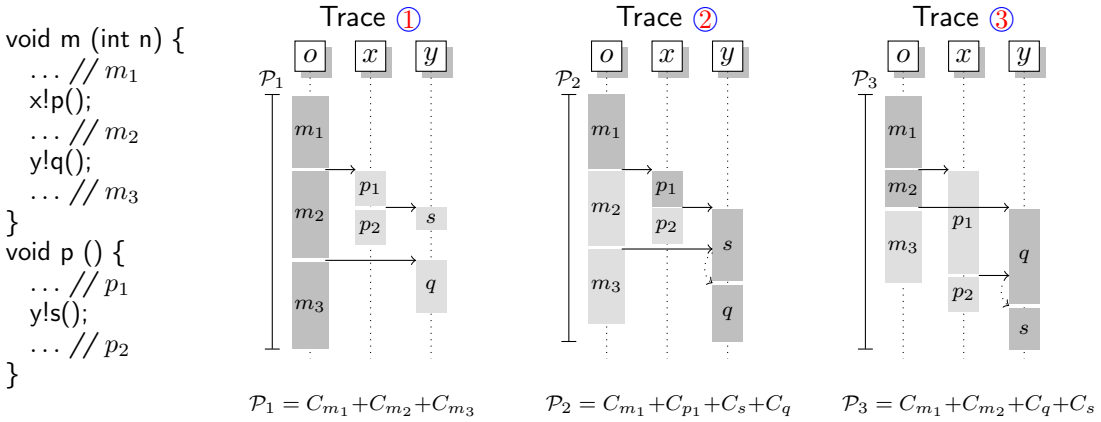


Figure 2.6: Parallel Cost Analysis example

be obtained by using a cost model that counts the number of times a point is visited within the analysis framework described in Section 2.1.

Example 21 *As observed in Example 20, C_{showCoin} bounds the execution of all such instances, but only one instance is queued simultaneously at object L37. If C^c_{showCoin} is the number of tasks spawned for method* showCoin*, $max(C_{\mathsf{showCoin}}/C^c_{\mathsf{showCoin}}, C_{\mathsf{showIncome}})$ is a more accurate bound for the peak cost executed by L37.*

The peak analysis presented in this section is integrated within the SACO tool delivered with this document. Section 3 contains a detailed explanation of its usage and how to visualize its results.

2.3.4 Parallel Cost Analysis

Parallel cost differs from the standard notion of *serial cost* by exploiting the truly concurrent execution model of distributed processing to capture the cost of synchronized tasks executing in parallel. It is also different to the peak cost since this one is still serial; i.e., it accumulates the resource consumption in each component and does not exploit the overall parallelism as it is required for inferring the parallel cost. It is challenging to infer parallel cost because one needs to soundly infer the parallelism between tasks while accounting for waiting and idle processor times at the different objects. Let us see an example.

Example 22 *Figure 2.6 (left) shows a simple method* m *that spawns two tasks by calling* p *and* q *at objects* x *and* y*, resp. In turn,* p *spawns a task by calling* s *at object* y*. This program only features distributed execution, concurrent behaviours within the objects are ignored for now. In the sequel we denote by C_m the cost of block m. C_{m_1}, C_{m_2} and C_{m_3} denote, resp., the cost from the beginning of* m *to the call* x!p()*, the cost between* x!p() *and* y!q()*, and the remaining cost of* m*. C_{p_1} and C_{p_2} are analogous. The resource analysis described in Section 2.1 can be used for obtaining an upper bound of the cost of each block.*

The notion of parallel cost $\mathcal{P}$ corresponds to the cost consumed between the first instruction executed by the program at the initial object and the last instruction executed at any object by taking into account the parallel execution of instructions and idle times at the different objects.

Example 23 *Figure 2.6 (right) shows three possible traces of the execution of this example (more traces are feasible). Below the traces, the expressions $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ show the parallel cost for each trace. The main observation here is that the parallel cost varies depending on the duration of the tasks. It will be the worst (maximum) value of such expressions, that is, $\mathcal{P}$=$max(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \dots)$. In ② p_1 is shorter than m_2, and* s *executes before* q*. In ③, q is scheduled before* s*, resulting in different parallel cost expressions. In ①, the processor of object* y *becomes idle after executing* s *and must wait for task* q *to arrive.*

In the general case, the inference of parallel cost is complicated because: (1) It is unknown if the processor is available when we spawn a task, as this depends on the duration of the tasks that were already in the queue; e.g., when task q is spawned we do not know if the processor is idle (trace ①) or if it is taken (trace ②). Thus, all scenarios must be considered; (2) objects can be dynamically created, and tasks can be dynamically spawned among the different objects (e.g., from object o we spawn tasks at two other objects). Besides, tasks can be spawned in a circular way; e.g., task s could make a call back to object x; (3) Tasks can be spawned inside loops, we might even have non-terminating loops that create an unbounded number of tasks. We use a *distributed flow graph* (DFG) to capture the different flows of execution that the distributed system can perform. We use the standard partitioning of methods into blocks to build the control flow graph of the program. The nodes in the DFG are the blocks of the control flow graph (CFG) combined with the object's identity and are used as cost centers when obtaining the upper bound as in Section 2.1. The edges represent the control flow in the sequential execution (drawn with normal arrows) and all possible orderings of tasks in the object's queues (drawn with dashed arrows) since, when the processor is released, any pending task of the same object could start executing.

Example 24 *Figure 2.7 shows the DFG for the program in Figure 2.6. Nodes in gray are the exit nodes of the methods, and it implies that the execution can terminate executing $o{:}m_3$, $x{:}p_2$, $y{:}s$ or $y{:}q$. Solid edges include those existing in the CFG of the sequential program but combined with the object's identity and those derived from calls. The dashed edges model that the execution order of s and q at object y is unknown.*

Our analysis consists of obtaining the maximal parallel cost from all possible executions of the program, based on the DFG. The execution paths in the DFG start in the initial node that corresponds to the entry method of the program, and finish in any exit node of a method. The key idea to obtain the parallel cost from paths in the graph is that the cost of each block contains not only the cost of the block itself but this cost is multiplied by the number of times the block is visited. For the sake of inferring the cost, it is not relevant the order in which blocks are executed and thus we use sets instead of sequences. The parallel cost of the distributed system can be over-approximated by the maximum cost for all paths to nodes that correspond to method exit blocks.

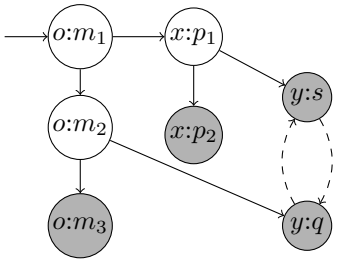


Figure 2.7: DFG for Figure 2.6

Example 25 *Given the DFG in Example 24, we have the following sets:*

$$\{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1}, \underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2}, \underbrace{\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\}}_{N_3}, \underbrace{\{o{:}m_1, o{:}m_2, y{:}s, y{:}q\}}_{N_4}\}$$

Observe that these sets represent traces of the program. The execution captured by N_1 corresponds to trace ① of Figure 2.6. In this trace, the code executed at object o leads to the maximal cost. Similarly, the set N_3 corresponds to trace ② and N_4 corresponds to trace ③. The set N_2 corresponds to a trace where $x{:}p_2$ leads to the maximal cost (not shown in Figure 2.6). The cost is obtained by using the block-level costs for all nodes that compose the sets above. The overall parallel cost is computed as:

$$max(C_{N_1}, C_{N_2}, C_{N_3}, C_{N_4}) \text{ where } \begin{array}{l} C_{N_1} = C_{o:m_1} + C_{o:m_2} + C_{o:m_3}, \\ C_{N_2} = C_{o:m_1} + C_{x:p_1} + C_{x:p_2}, \\ C_{N_3} = C_{o:m_1} + C_{x:p_1} + C_{y:s} + C_{y:q}, \\ C_{N_4} = C_{o:m_1} + C_{o:m_2} + C_{y:s} + C_{y:q} \end{array}$$

Importantly, the parallel cost is more precise than the serial cost because all paths have at least one missing node. For instance, N_1 does not contain the cost of $x{:}p_1$, $x{:}p_2$, $y{:}s$, $y{:}q$ and N_3 does not contain the cost of $o{:}m_2$, $o{:}m_3$, $x{:}p_2$.

```
1 Unit main(Int n){
2   while(n>0){
3     Job j=new Job();
4     j!start(10);
5     await duration(1,1);
6     n= n−1;
7   }
8 }
```

```
9 class Job{
10  Unit start(Int dur){
11    while(dur>0){
12      [Cost: 1] dur=dur−1;
13      await duration(1,1);
14    }
15  }
16 }
```

Figure 2.8: Example of timed model

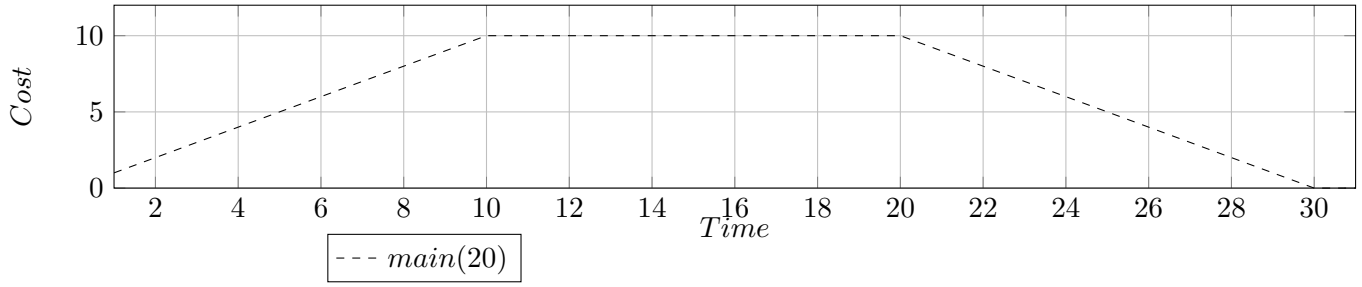


Figure 2.9: Resource consumption in time of the example in Fig.2.8

2.3.5 Cost Analysis in Time

With the introduction of concurrency and distribution, the concept of resource consumption or cost becomes much richer that in the traditional sequential non-distributed setting. Instead of having a single magnitude, we can have a cost for each distributed component (using cost centers) and other cost measures such as peak cost or parallel cost (See Sections 2.3.3 and 2.3.4).

One of the possibilities that ABS offers is to extend its models with time annotations [21]. These time annotations model the passing of time during the execution of the model and add a new dimension to the cost of a model. Given an ABS model with time annotations, we want to obtain upper bounds on the resource consumption at any given time. Instead of having a simple measure of how many resources are needed to execute a model, we can obtain a profile of how many resources are needed at each time interval and how the resource requirements evolve with the passage of time.

Example 26 *The code in Figure 2.8 represents a simple model with time annotations (Lines 5 and 13). The main method launches n jobs of duration* 10 *(Line 4). However, the jobs are not launched simultaneously but one each time unit. The jobs take* 10 *time units to complete and consume one cost unit at a time (Line 12).*

In Figure 2.9 we can see the cost in time of the example for $n = 20$*. The number of running jobs grows steadily from* 1 *to* 10*. At that point, the jobs launched at the beginning start to finish but new jobs keep being added until the time is* 20*. After time* 20*, no more jobs are created and the existing ones finish one by one until the time is* 30*.*

SACO can generate the following piece-wise upper bound that precisely represents the model's behavior:

Upper Bound	*Precondition*
0	$(t_o = n + 1 \wedge t_o >= 2 \wedge t >= t_o + 9) \vee (t_o = 1 \wedge 0 >= n \wedge t >= 1)$
1	$(t = 1 \wedge t_o = n+1 \wedge t_o >= 3) \vee (t_o = n+1 \wedge t_o+8 = t \wedge t_o >= 2) \vee (n = 1 \wedge t = 1 \wedge t_o = 2)$
10	$(t_o = n+1 \wedge t >= 11 \wedge t_o >= t+2) \vee (t = 10 \wedge t_o = n+1 \wedge t_o >= 12) \vee (n = t \wedge n+1 = t_o \wedge n >= 11) \vee (n = 10 \wedge t = 10 \wedge t_o = 11)$
n	$(t_o = n + 1 \wedge 9 >= t \wedge t_o >= 2 \wedge t >= t_o) \vee (t = 10 \wedge n + 1 = t_o \wedge 9 >= n \wedge n >= 2)$
t	$(t_o = n + 1 \wedge 9 >= t \wedge t >= 2 \wedge t_o >= t + 2) \vee (n = t \wedge n + 1 = t_o \wedge 9 >= n \wedge n >= 2)$
$n + 10 - t$	$(t_o = n + 1 \wedge t >= 11 \wedge t_o + 7 >= t \wedge t >= t_o)$

n is the entry parameter, t the time we want to observe and t_0 the final time after the execution of method main. If we evaluate this upper bound for $n = 20$ and for the interval $t = [1..30]$, we obtain the graph in

Example 1	Example 2	Example 3

while(0<i && i<n){
if(dir==1)
i=i+1;
else
i=i−1;
}
	while(x>0 && y>0){
	if(∗){
	x=x−1;
	y=r;
	} **else**{
	y=y−1;
	}
	}
		while(x>0){
		while(y>0 && ∗){
		y=y−1;
		}
		x=x−1;
		}

Figure 2.10: Example of complex loops

Figure 2.8.

In order to analyze timed models, we developed a model transformation that takes a model with time primitives and that generates a regular ABS model whose cost is equivalent to the cost of the original model at time t. In this transformed model t is simply an additional input parameter of the model. Once the model has been transformed, we can apply a regular cost analysis to obtain the corresponding upper bounds. In order to obtain high precision, it might be necessary to use the CoFloCo backend described in Section 2.4. In particular if we want to obtain piece-wise upper bounds such as the one given in example 26.

This capability to analyze timed ABS models has still some limitations. Only models with explicit cost annotations without *blocking* instructions are supported. An instruction is blocking if it can prevent a complete COG from progressing (such as y.**get** instructions that are not preceded by the corresponding **await** y?). For more details see Appendix I.

The timed analysis presented in this section is integrated within the SACO tool delivered with this document. Section 3 contains a detailed explanation of its usage.

2.4 CoFloCo: a more precise backend

The cost analyses in SACO follow a two phase approach. In the first phase, a set of cost equations is generated. Cost equations are recursive equations annotated with constraints that represent the cost of the target program. In the second phase these cost equations are analyzed and a closed-form upper bound is generated. That is, an upper bound that can be directly evaluated. This second phase is performed by PUBS [5] and is common to all cost analyses of SACO. Solving cost equations into a closed form is a complex process and PUBS fails to obtain (precise) upper bounds for many problems. Because of this, we developed an alternative backend CoFloCo (Appendix J) that can obtain (more precise) upper bounds for many programs where PUBS fails. Because of having a compatible interface, CoFloCo has been directly integrated in the SACO tool as an option so all the analyses can benefit for its increased precision. Section 3 contains a detailed explanation of its usage.

CoFloCo incorporates multiple techniques to achieve higher precision. In Figure 2.10, we illustrate through examples some of the cases where CoFloCo can obtain better bounds than PUBS:

Ex1 We have a loop that finishes when i reaches 0 or n and that can increase or decrease i in its body. i is guaranteed to reach 0 or n because whether i is incremented or decremented depends on a condition that does not change throughout the execution. That is, once entered in the loop, always the same path in the loop will be taken. That leaves us with two possible execution patterns. One in which $dir = 1$ and the loop iterates $n - i$ times and another one in which $dir \neq 1$ and the loop iterates i times. CoFloCo is able to analyze these two patterns separately and thus find a bound $max(n - i, i)$ whereas PUBS does not find any upper bound.

Ex2 The loop decrements either x or y in each iteration given a condition that we consider abstracted away $*$. However, when x is decremented, y is reset to a value r. The first path of the loop can be executed x

times and the second path can be executed $y + (x * r)$ times which account to y being set to r x times. In total the upper bound of the loop in the number of iterations is $x + y + (x * r)$. PUBS consider all the loop paths together and fails to obtain any bound.

Ex3 If we count the number of times y is modified, PUBS obtain an upper bound $x * y$. This upper bound is correct but imprecise. Because y is not reset every time the inner loop is reached, it can be easily seen that y is a valid and more precise upper bound despite having two nested loops. This bound is the one obtained by CoFloCo.

Traditionally, cost analyses obtain a single upper bound function valid for all possible input values. This information can be insufficient in cases where the behavior of a program or model depends heavily on the values of the input parameters. CoFloCo can obtain a set of *conditional upper bounds*. Conditional upper bounds are upper bounds that are valid as long as a precondition is satisfied. This preconditions are generated based on the internal structure of the program. If we combine the different conditional upper bounds we can generate a piece-wise upper bound function that provide us with more detailed information of the behavior of the analyzed program.

Example 27 *In the example 1 from Figure 2.10, we can obtain the following piece-wise upper bound:*

Upper Bound	*Precondition*
$n - i$	$dir = 1 \wedge i \geq 1 \wedge i \leq n - 1$
i	$dir \neq 1 \wedge i \geq 1 \wedge i \leq n - 1$
0	$(i \leq 0) \vee (i \geq n)$

Chapter 3

End User Documentation

3.1 User Manual

In what follows we present the user manual of SACO by illustrating the different analyses that it is able to perform. In Figure 3.2, we first show how to start to use SACO within the Envisage collaboratory. For this, we must first select the analysis in a pull-down menu, and, for executing the analysis, we click on Apply. The Clear button removes all previous results. Optionally, the parameters of the selected analysis can be configured by clicking on Settings (details are given in Section 3.1.1). Finally, the results of the selected analysis are presented in the console. This can be done by means of graphs, text messages, markers, highlighters in the code, and interactions among them. In the following, we describe the use of SACO by analyzing the examples studied throughout Section 2.2.

3.1.1 Parameters of the Analyses

Basic Resource Analysis

Let us start by performing the basic resource analysis described in Section 2.1. We open the file VendingMachine_init.abs, which contains the code described in Example 1 and that can be found within the DeliverableExamples project. Note that in Example 1 we had to ignore the await instruction at L26 in the code of Figure 2.1. Thus, it is commented in the file VendingMachine_init.abs. We click on Refresh Outline and we select the entry method (method main of class IMain) in the Outline (see right of Figure 3.1). By clicking Settings a pop-up window appears and shows the configuration that allows us to set up the parameters for the analysis. Let us select Resource Analysis and the following parameters can be configured (see Figure 3.3):

Cost model: The cost model indicates the type of resource that we are interested in measuring. The user can select among the following cost metrics: termination (only termination is proved), steps (counts the number of executed instructions), objects (counts the number of executed new instructions), tasks (counts the number of asynchronous calls to methods), memory (measures the size of the created data structures), data transmitted (measures the amount of data transmitted among the distributed objects), user-defined model (allows to write annotations in the code of the form [cost == expr] and the analysis accumulates the cost specified by the user in expr every time this program point is visited).

Cost centers: This option allows us to decide whether we want to obtain the cost per cost center (i.e., for each of the abstract objects inferred by the analysis) or a monolithic expression that accumulates the whole computation in the distributed system. The value no refers to the latter case. If we want to separate the cost per cost center, we have again two possibilities. The option class shows the cost of all objects of the same class together, while object indicates the cost attributed to each abstract object.

Asymptotic bounds: Upper bounds can be displayed in asymptotic or non-asymptotic form. The former one is obtained by removing all constants and subsumed expressions from the non-asymptotic cost, only showing the complexity order.

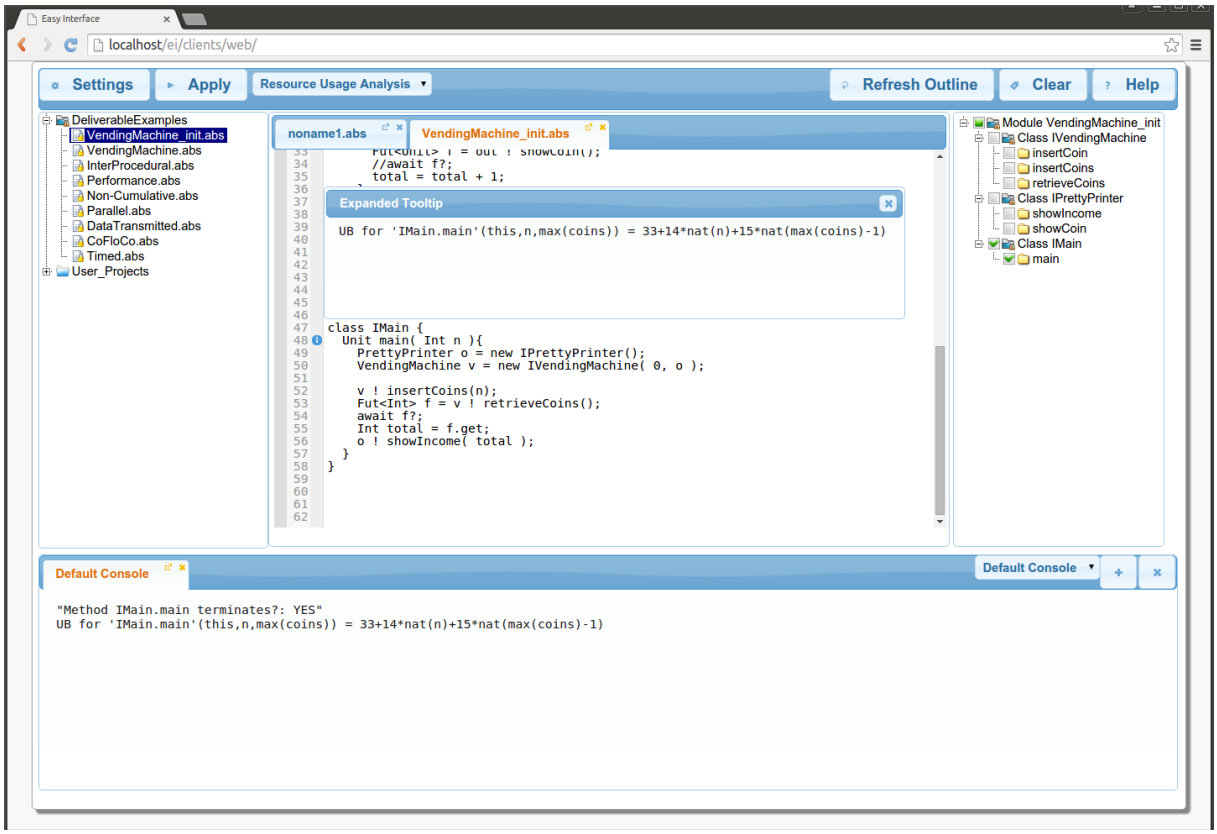


Figure 3.1: Collaboratory web Interface

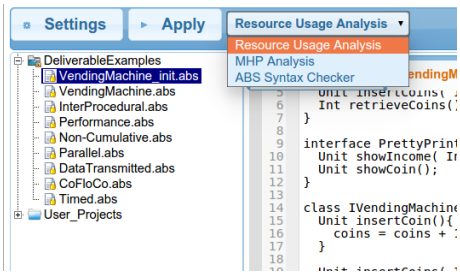


Figure 3.2: Collaboratory web interface analysis selection

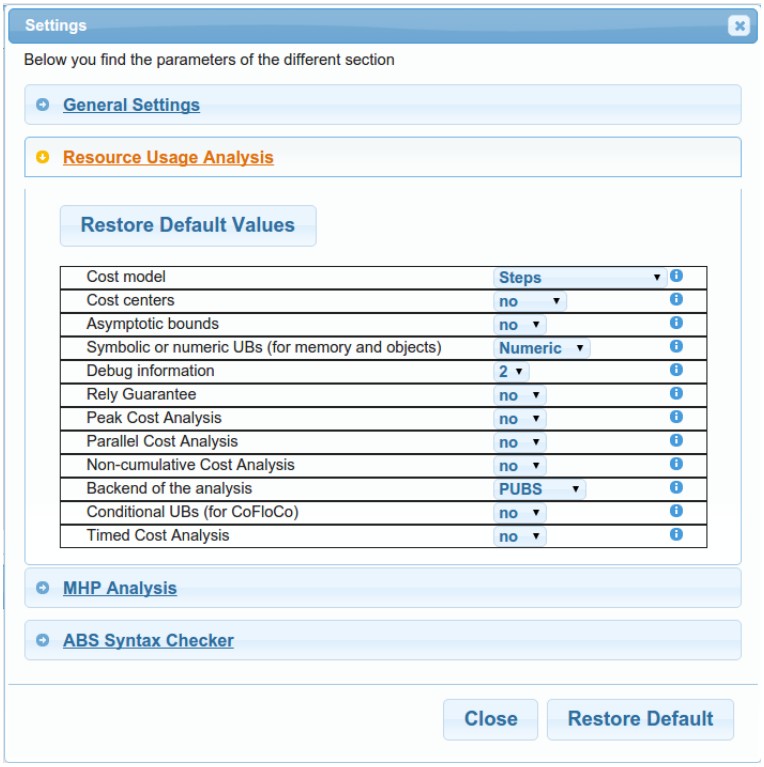


Figure 3.3: Resource analysis options in SACO

Symbolic or numeric: Next, if the cost model is memory or objects, the upper bounds can be shown either, *symbolically*, in terms of symbolic sizes (we use size(A) to refer to the size of an object of type A), or *numeric*, by assigning a predefined measure to them.

Debug: sets the verbosity of the output (the higher the number, the more verbose the output).

Rely Guarantee: performs the resource analysis taking into account the possible interleavings in the tasks execution (see Section 2.2.2).

Peak Cost Analysis: computes the peak cost analysis for all objects which are identified (see Section 2.3.3).

Parallel Cost Analysis: computes the parallel cost analysis of the program (see Section 2.3.4).

Non-cumulative Cost Analysis: computes the non-cumulative cost of the program (see Section 2.3.2).

Backend of the Analysis: SACO uses *PUBS* or *CoFloCo* as backend to solve the cost equations (see Section 2.4).

Conditional UBs: computes a set of *conditional upper bounds* according to some conditions on the input parameters (see Section 2.4).

Timed Cost Analysis: computes the cost analysis in time (see Section 2.3.5).

Let us analyze the program VendingMachine_init.abs with the default values, except for the asymptotic bounds parameter that we set to yes. We click on Close and then we click on Apply. Figure 3.4 shows the resource analysis results yield by SACO. It can be seen that the upper bound is linear and it is a function on n (the input parameter of main) and on the maximum value that field coins can take, denoted max(coins). Variable n is wrapped by function nat previously defined to avoid negative costs. The upper bound is shown in the console view and also at the method's header when the mouse passes over the marker (see L48 in Figure 3.4).

Now, let us analyze the main method of the file VendingMachine.abs which contains the code shown in Figure 2.1—including the await instruction at L26. The results of the analysis can be seen in Figure 3.5, and,

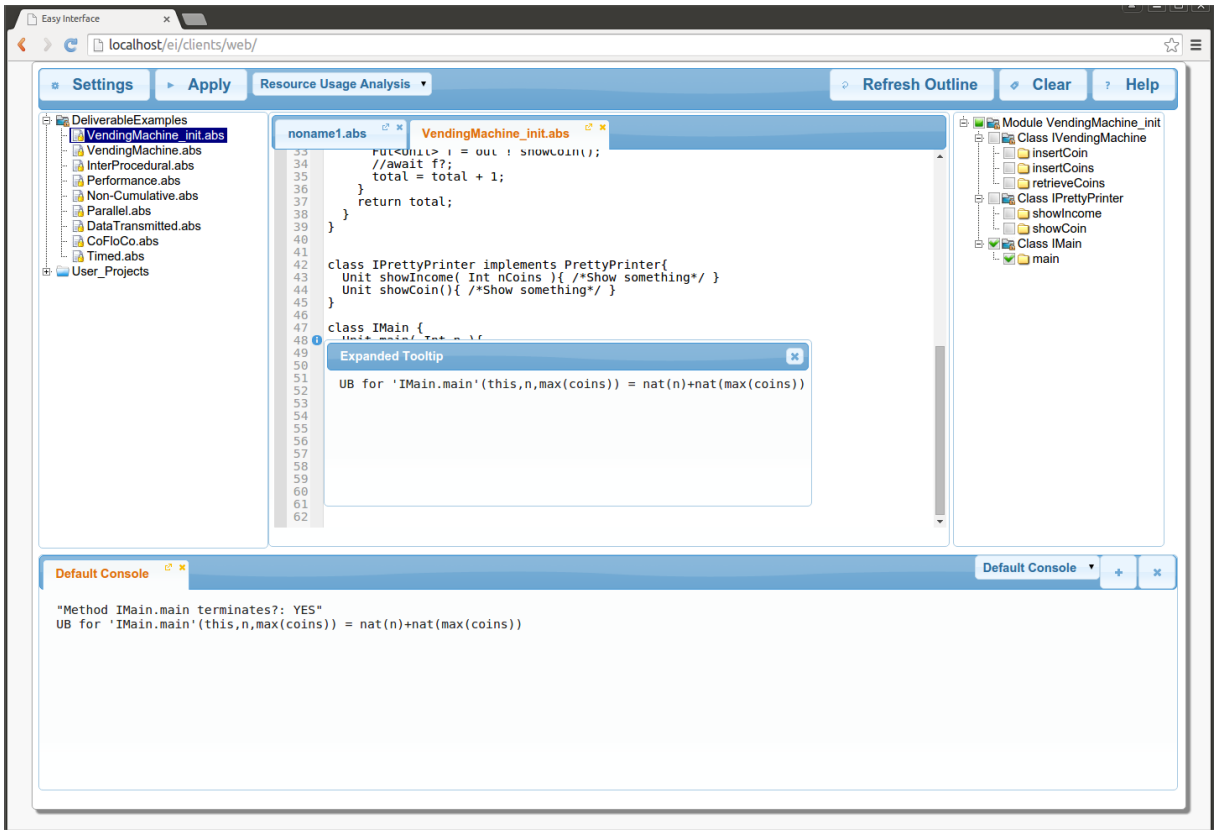


Figure 3.4: SACO's Starting Point Resource Analysis

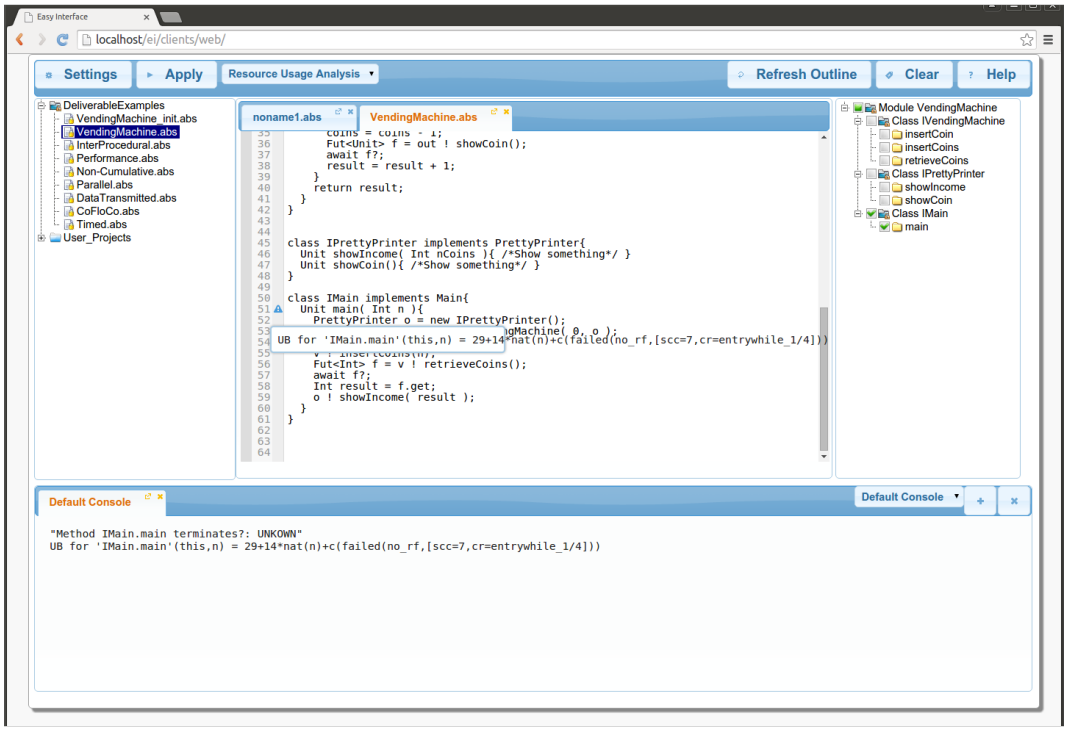


Figure 3.5: Problem with the basic resource analysis

as we have explained in Example 3, it shows, by using a warning marker, that the resource analysis cannot infer an upper bound nor guarantee the termination of the program.

Rely-guarantee Resource Analysis

Let us now perform the analysis described in Section 2.2.2 on the main of the VendingMachine.abs file. To do so, we set the option Rely Guarantee to yes and the Cost Model to termination.

Figure 3.6 shows that SACO proves all methods of the example of Figure 2.1 terminating. Let us now slightly modify the example to make method insertCoins non-terminating by removing the line with the instruction coins = coins - 1. The analysis information is displayed as follows. For each *strongly connected component*[1] (SCC), the analysis places a marker in the entry line to the SCC. If the SCC is terminating, by clicking on the marker, the lines that compose this SCC are highlighted in yellow (see Figure 3.6). On the other hand, if the SCC is non-terminating, by clicking on the marker (see L29 of Figure 3.7) SACO highlights the lines of the SCC in blue. Besides the markers, the list of all SCCs of the program and their computed termination results are printed by SACO in the console.

At this point, let us perform the rely guarantee resource analysis to infer the cost of the program. We select the Steps cost model with the option Rely guarantee set to yes. Figure 3.8 shows the computed upper bound. The upper bound is a function on n (the input parameter of main) multiplied by the maximum value that field coins can take, denoted max(coins). We can observe that the cost of main is quadratic. In addition, SACO shows a marker for each method to display their corresponding upper bounds.

May-Happen-in-Parallel with Inter-Procedural Synchronization

Let us analyze with SACO the program shown in Figure 2.2 with the inter-procedural synchronization of the MHP analysis, as we have described in Section 2.2.3. To do so, we select the May-Happen-in-Parallel analysis and set the option Inter-Procedural Synchronization to yes. Figure 3.9 shows the output of SACO

[1] While loops and methods are basically the SCCs in a program.

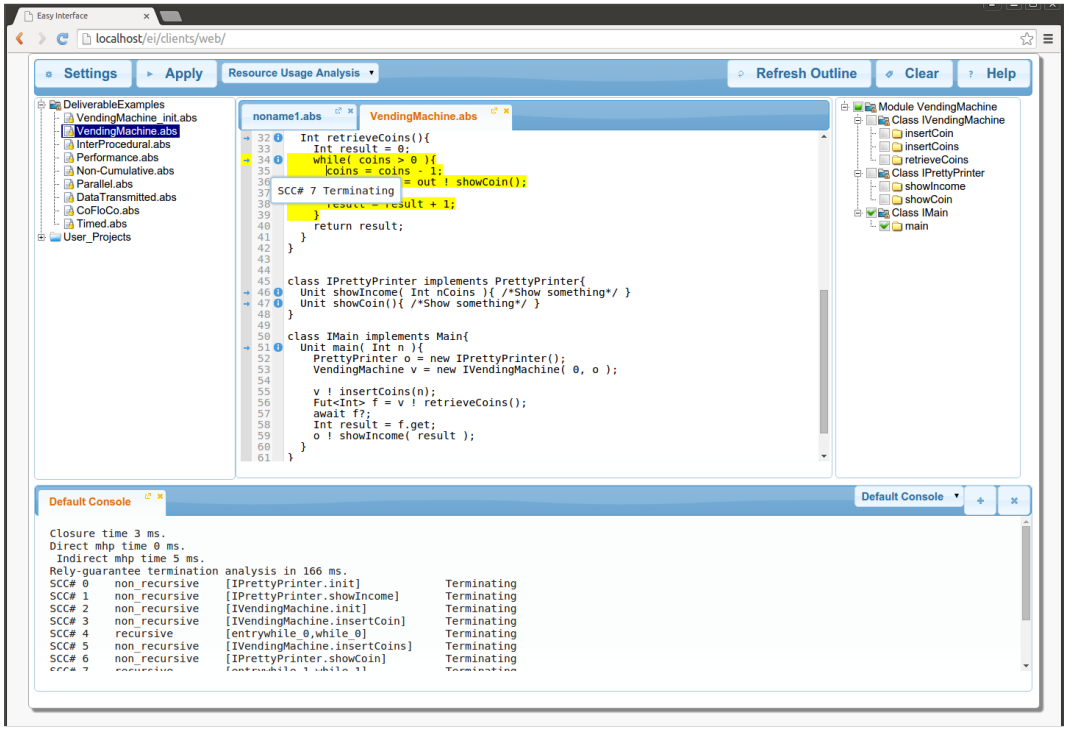


Figure 3.6: Rely guarantee termination analysis output of SACO for the example in Figure 2.1

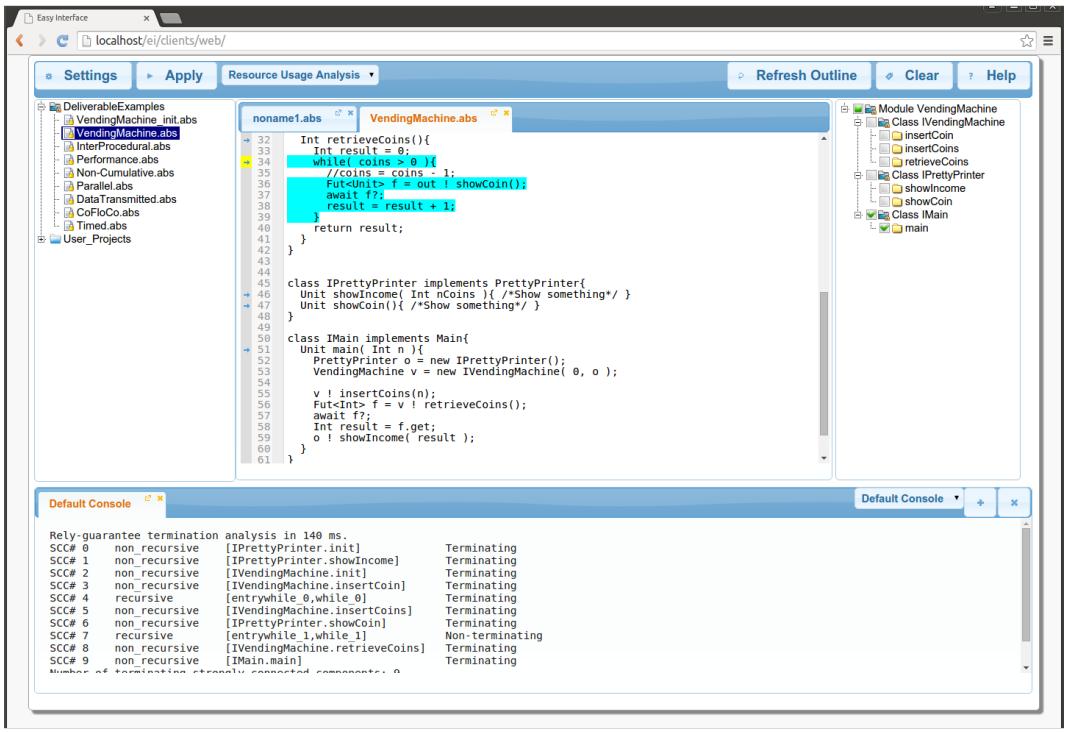


Figure 3.7: Rely Guarantee Termination analysis results for a non-terminating example

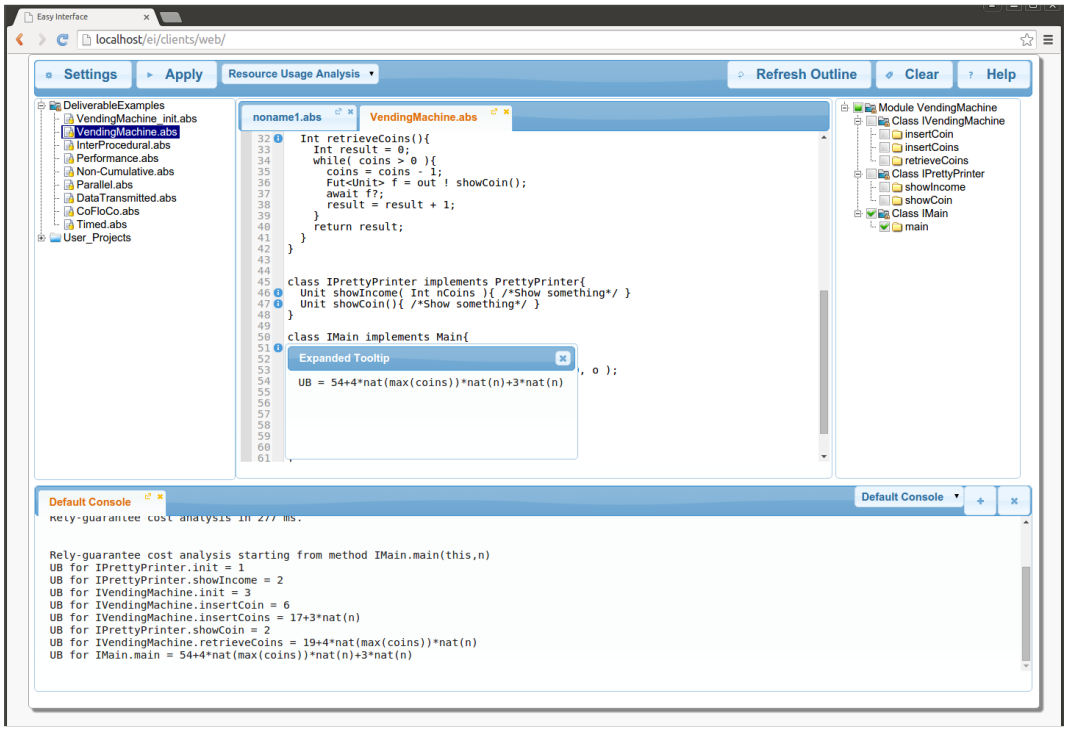


Figure 3.8: Rely guarantee resource analysis results for the example in Figure 2.1

after analyzing method main. By clicking in the blue arrows that appear to the right of the line numbers, SACO highlights those program points that might happen in parallel with the clicked program point. In Figure 3.9 we see that the only program points that can happen in parallel with L34 are the end of methods f and g, which means that both methods must have finished when L34 is reached.

Load Balance

At this point, let us use the resource analysis to study the load balance of the example used in Section 2.3.1. First, we open the file LoadBalance.abs in the project DeliverableExamples. We start by applying the Resource Analysis and selecting the option Cost Centers to object. As the concurrency unit of ABS is the object, it uses the cost centers to assign the cost of each execution step to the object where the step is performed. If we apply the Resource Analysis to the method C.m, SACO returns that we have two cost centers, one cost center labelled with [12] which corresponds to the object that executes C.m and [13, 12], which abstracts the object created at $L13$. The labels of the nodes contain the program lines where the corresponding object is created. That is, the node labeled as [13, 12] corresponds to the C object, created at L13 while executing the main method, node identified by L12. In addition, SACO shows a graph with both nodes in the Console Graph view. By clicking on the node [12], SACO shows a dialog box with the upper bound on the number of steps performed by such node. Similarly, by clicking on the node [13, 12], it shows the number of steps that can be executed by the object identified with [13, 12]. If we analyze method C.mthis, which contains the modification described in Figure 3.7, the cost is distributed differently, as we have detailed in Figure 3.10. Such differences can be seen by comparing the expressions shown in Figure 3.10 and those in Figure 3.11.

Then, to obtain the number of instances of each object we can have in C.m2, we perform the Resource-Analysis setting the options Cost Model to Objects and Cost Centers to Object. Figure 3.12 shows the output of SACO for this analysis, and it can be seen that, the number of instances of the object identified by [37, 35] is bounded by n (the input argument of method mthis). Finally, we can apply the ResourceAnalisis to C.m2 selecting Cost Model to Steps and its results are shown in Figure 3.13.

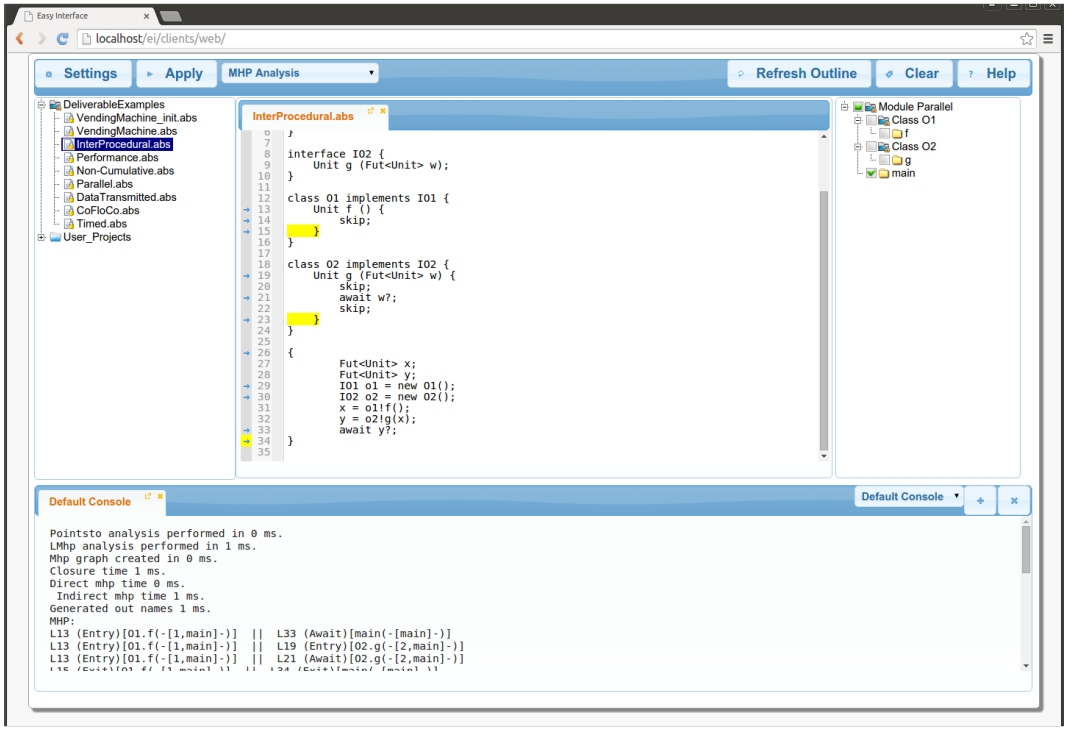


Figure 3.9: Inter-Procedural MHP analysis results for the example in Figure 2.2

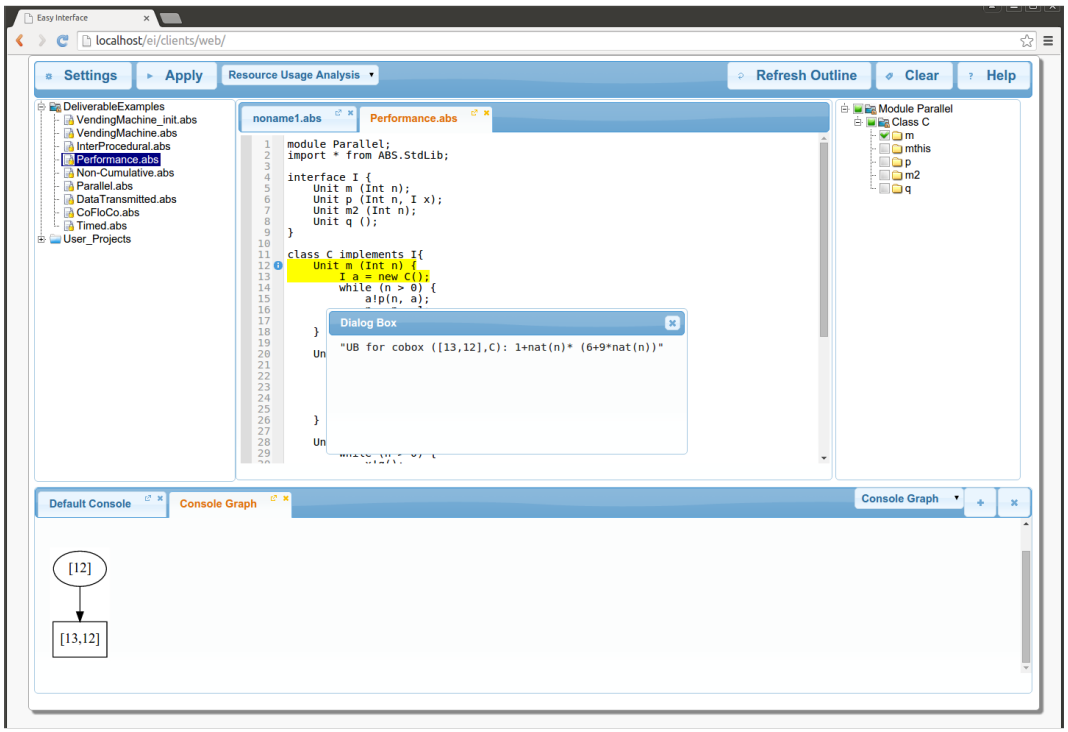


Figure 3.10: Resource Analysis to get the load balance for method C.m of the program in Figure 2.3

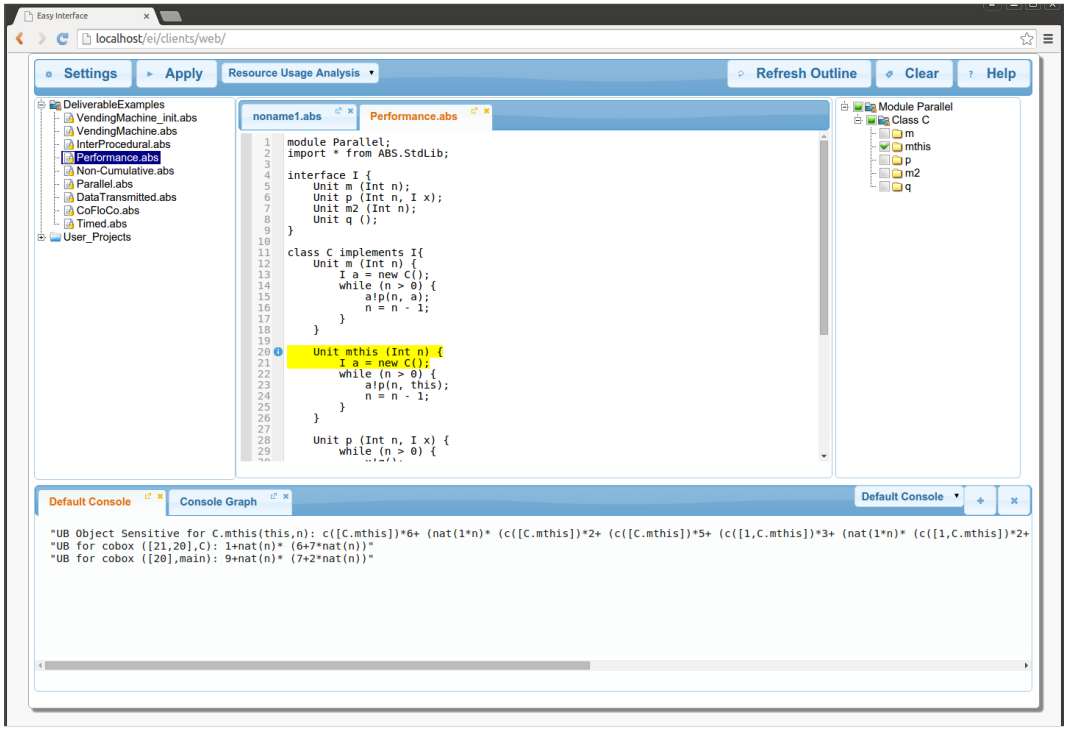


Figure 3.11: Resource Analysis to get the load balance for method C.mthis of the program in Figure 2.3

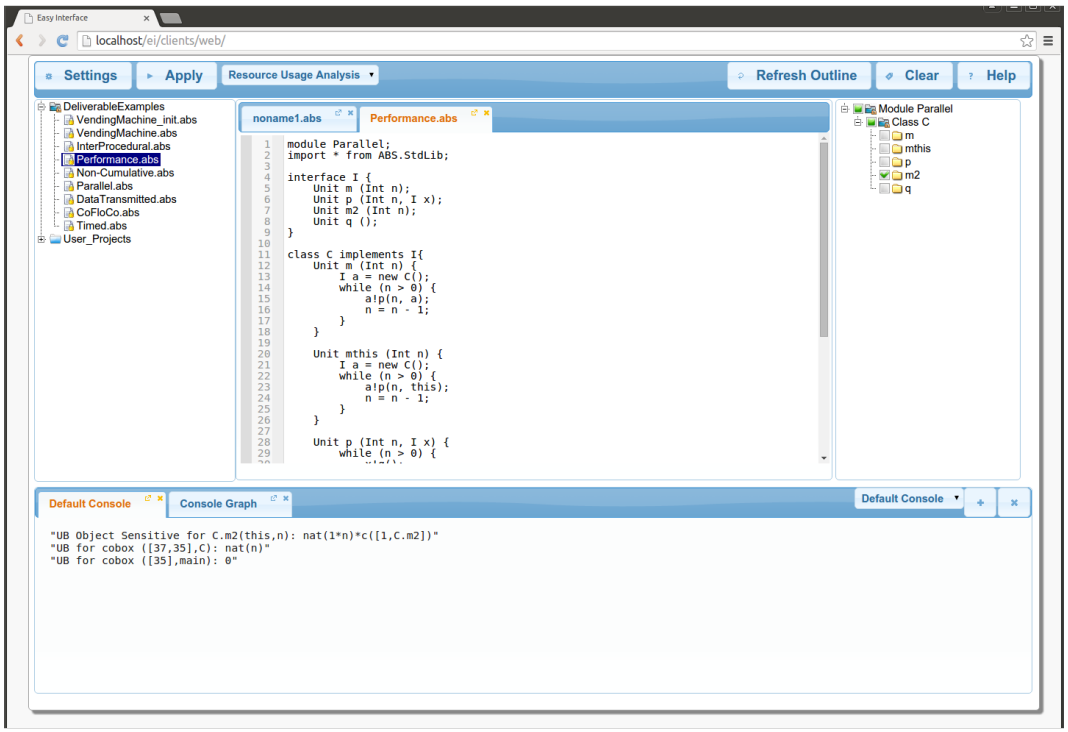


Figure 3.12: Resource Analysis to get the number of instances created by C.m2 of the program in Figure 2.3

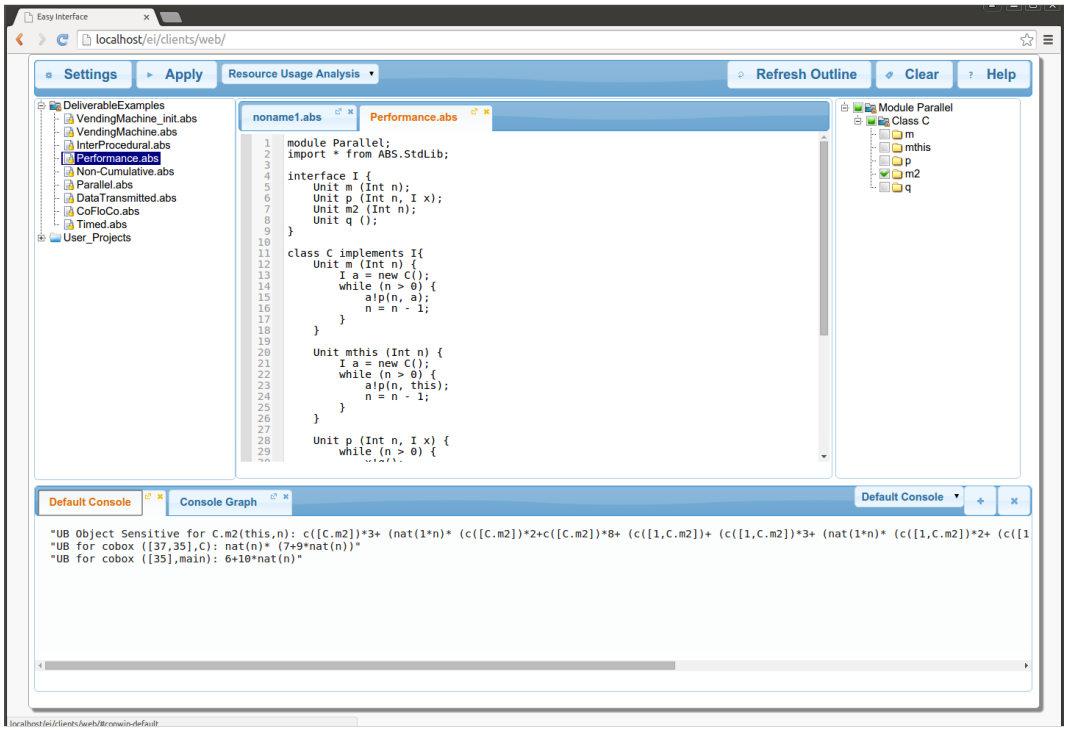


Figure 3.13: Resource Analysis to get the load balance for method C.m2 of the program in Figure 2.3

Transmission data sizes

Now, let us perform the transmission data size analysis for the example shown in Figure 2.4. First, we open the file DataTrasmitted.abs in the project DeliverableExamples. The transmission data size analysis is performed by selecting the analysis Resource Analysis and setting the option Cost Model to Data Transmitted. If we apply the analysis to the method IMain.main of the file DataTrasmitted.abs, the console will show the upper bound expressions for all possible pairs of objects identified by the analysis. In addition to the console information, a graph that shows the objects creation is displayed. The Console Graph is shown next to the default console. By clicking on a node, a message outputs the UBs for all transmissions data sizes that the selected object can perform and the objects involved in such transmissions. E.g., by clicking on the node $[32, 31]$, which corresponds to the Master object, we can see the upper bounds on the data transmitted (incoming and outgoing transmissions) from this object. As before, the labels of the nodes contain the program lines where the corresponding object is created. For instance, the node labeled as $[32, 31]$ corresponds to the Master object, created at L32 while executing the main method, the object identified by L31. This can be seen in Figure 3.14. In such upper bounds, the cost expression c(i) represents the cost of establishing the communication, that is, $\mathcal{I}$ explained in Example 17.

Non-cumulative Cost

The file Non-Cumulative.abs in the project DeliverableExamples contains the program shown at Figure 2.5. Let us perform the Resource Analysis by setting the option Cost Center to Non-cumulative to method IMain.main after restoring the default options. Figure 3.15 shows the results obtained by SACO. Such results show that we have two sets of program points that can lead to the maximum on the number of steps and their corresponding upper bound expressions. The set $[L6, L8, L10]$ corresponds to the acquire instructions at lines L6, L8 and L10 of the program.

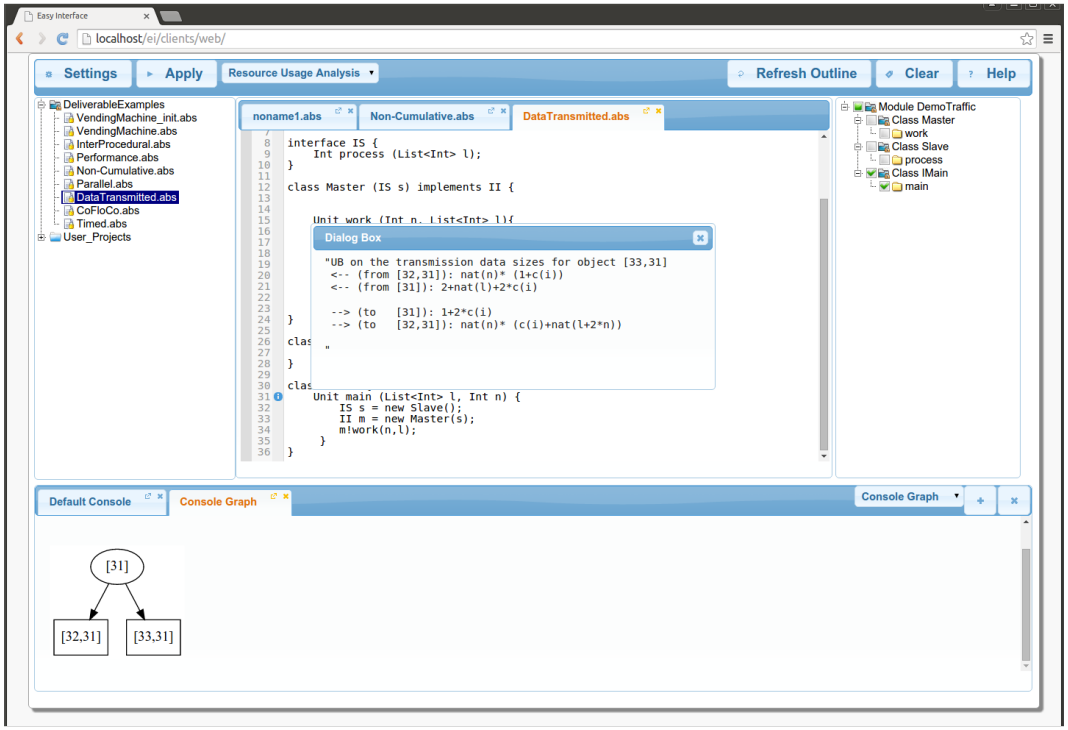


Figure 3.14: Transmissions data sizes analysis results for Example 2.4

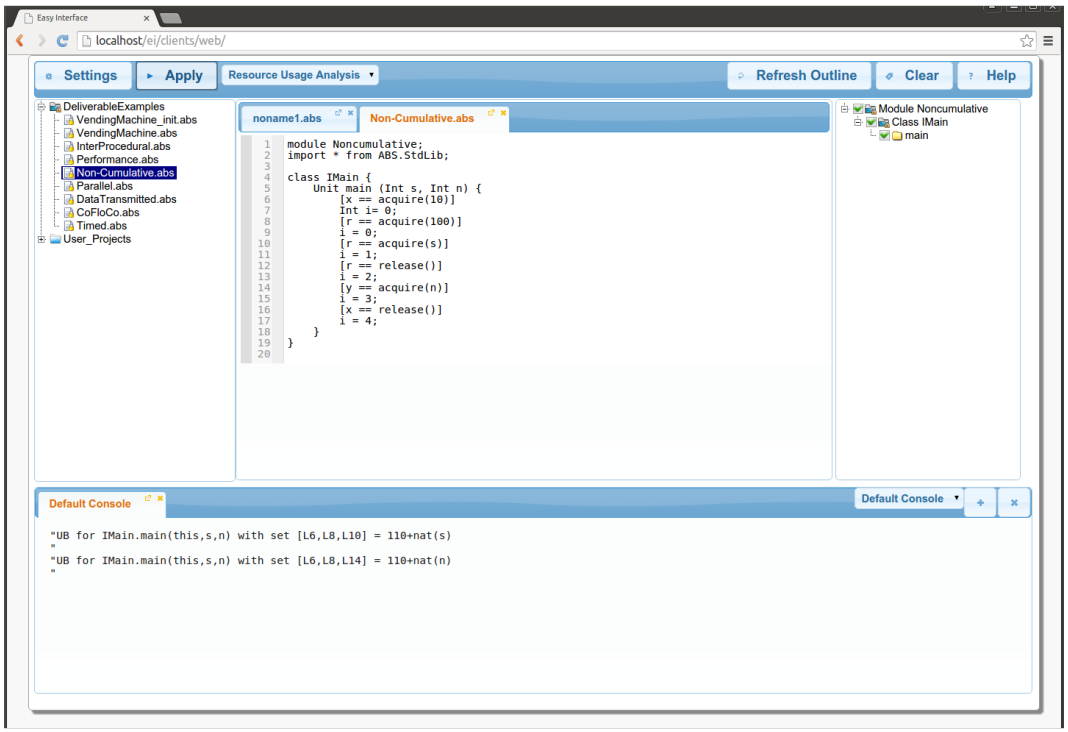


Figure 3.15: Non-Cumulative Resource Analysis for the program in Figure 2.5

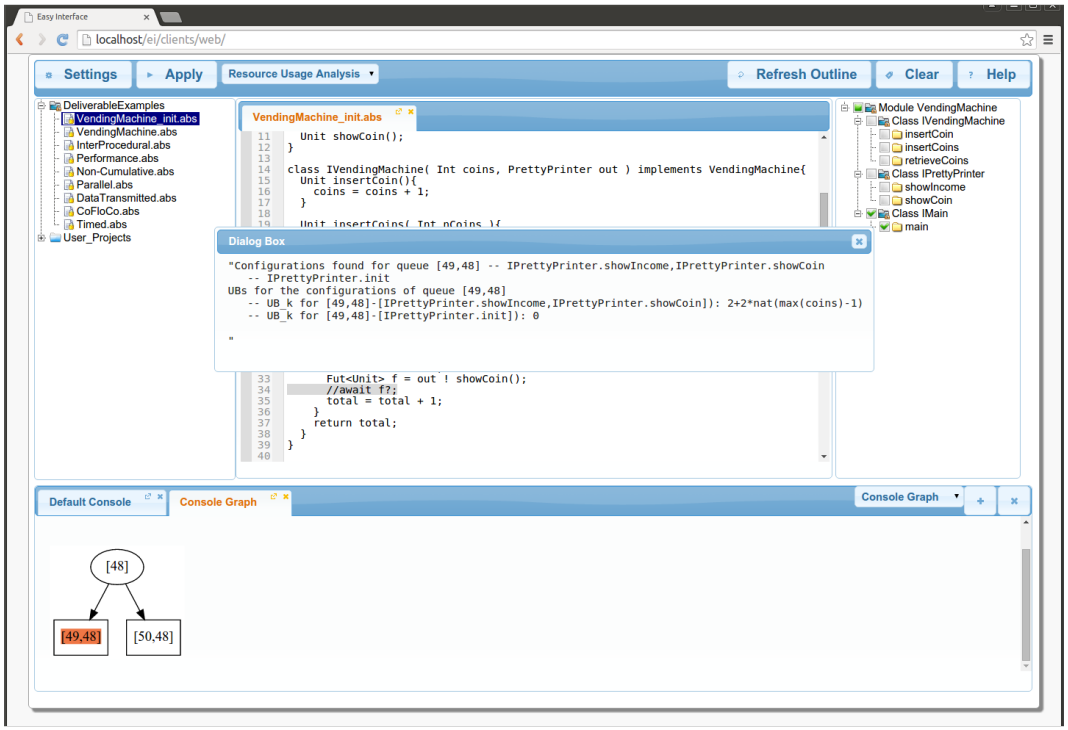


Figure 3.16: Peak cost analysis results for the example in Figure 2.1

Peak cost analysis

Let us continue by performing the peak cost analysis to VendingMachine_init.abs. Similarly to other analyses, we first select the entry method (method main in class IMain) in the Outline View and apply the Resource Analysis with the default options with the exception of the option Peak Cost Analysis, which is enabled to yes. For each identified object, the peak cost analysis outputs in the console all possible queue configurations and the cost associated to each of them. As before, SACO shows a graph where the labels of the nodes contain the program lines where the corresponding object is created. For instance, the graph shown in Figure 3.16, the node labeled as $[49, 48]$ corresponds to the PrettyPinter object, created at L49 while executing the main method (L48). By clicking on a node, the queue configurations that have been identified and their costs are shown in a message. The analysis results that are yield for the peak cost of PrettyPrinter can be seen in Figure 3.16.

Parallel Cost

Let us perform the parallel cost analysis that we have described in Section 2.3.4. To do so, we open the file Parallel.abs file of the project DeliverableExamples. Now, we select the entry method IMain.main in the Outline and apply the Resource Analysis by restoring the default values and setting the Parallel Cost Analysis to yes. Figure 3.17 shows the computed upper bound expressions obtained for all paths identified in the DFG of the program. In addition, SACO shows the number of nodes and edges of the computed DFG.

Cost Analysis in Time

Let us continue by performing the cost analysis in time to the program in Timed.abs. We select the entry method IMain.main in the Outline of the program. Then, we select the Resource Analysis and set the option Timed to yes. Figure 3.18 shows the output of SACO, showing the results detailed in Example 26 for the application of the cost analysis in time to the model shown in Figure 2.8.

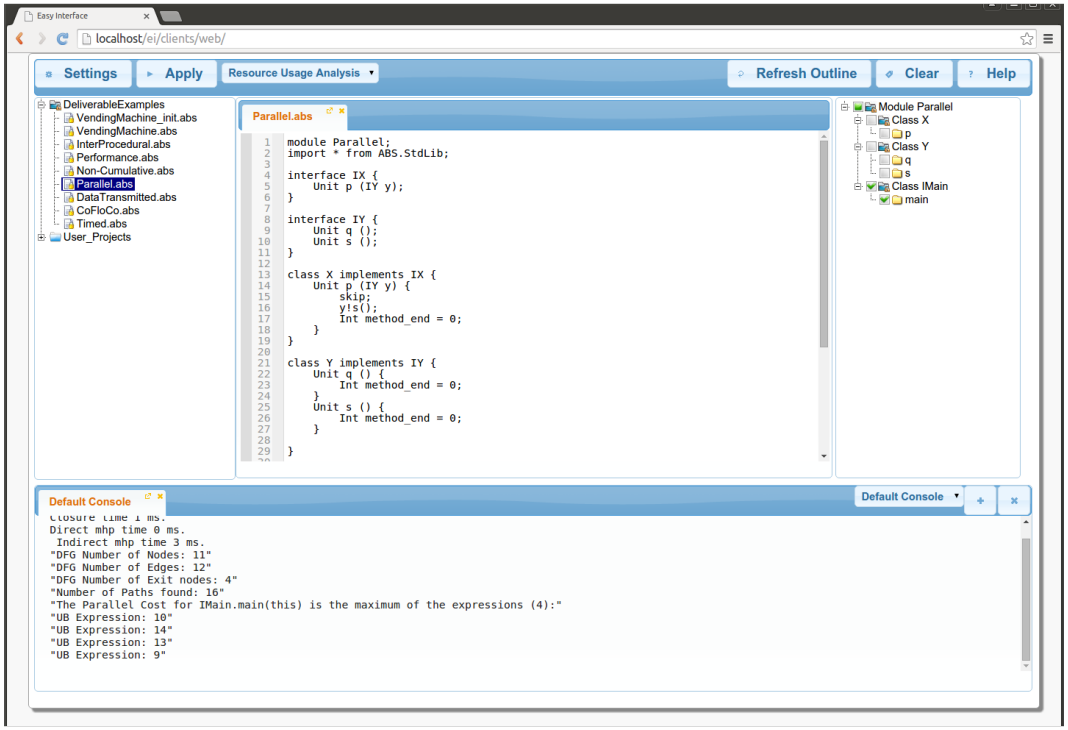


Figure 3.17: Parallel Cost analysis for the example in Figure 2.6

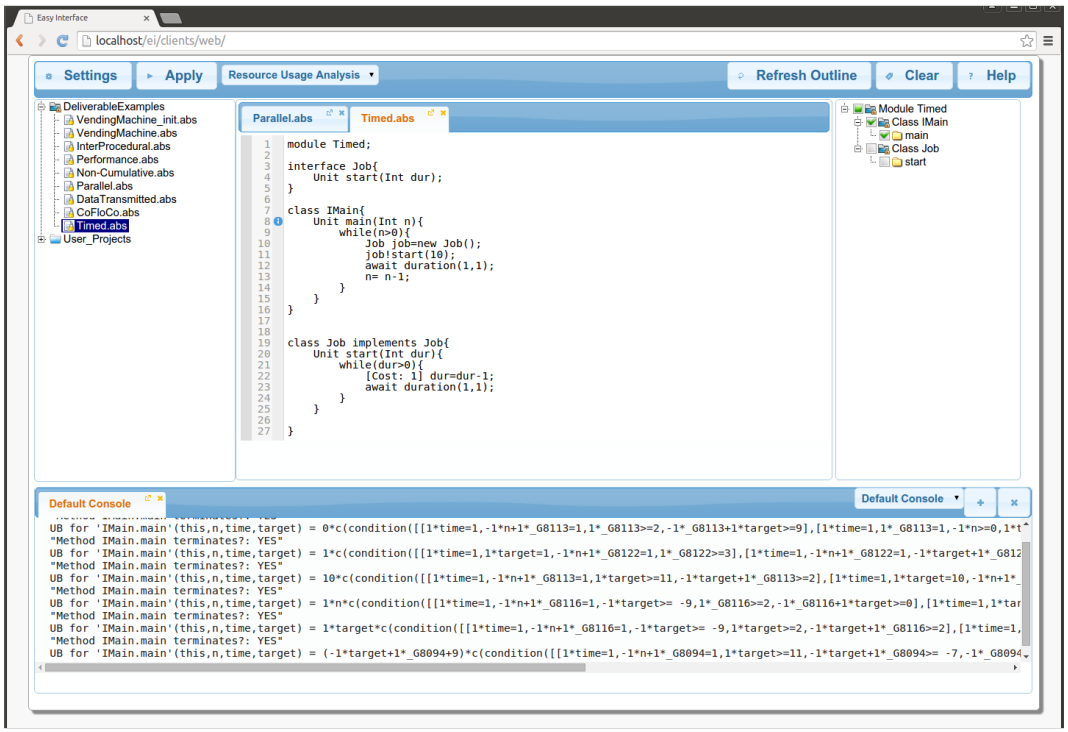


Figure 3.18: Cost analysis in time analysis output for the example in Figure 2.8

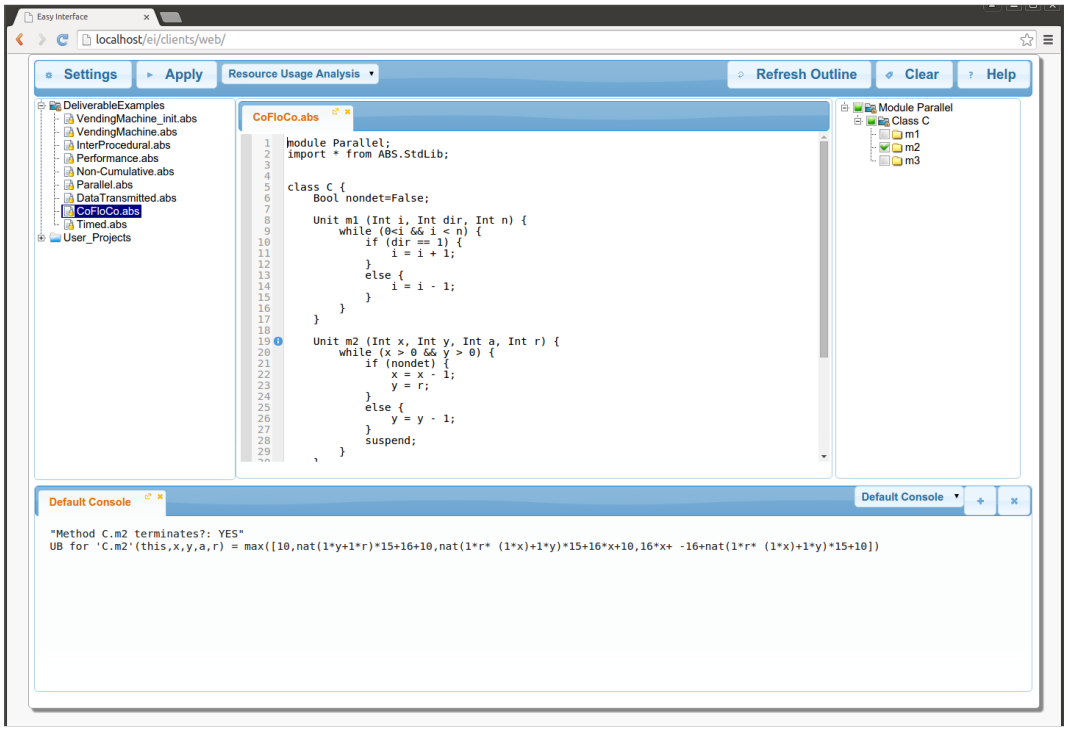


Figure 3.19: Resource analysis output for method C.m2 in the example of Figure 2.10 using CoFloCo

CoFloCo Backend

Finally, let us analyze the program CoFloCoExample.abs by using CoFloCo as backend. We can select the method of interest, that is, C.m, C.m2 or C.m3 and then perform the Resource Analysis with the default options except the option Backend, which must be set to CoFloCo. Figure 3.19 shows the results of the analysis of C.m with CoFloCo. Additionally, by setting the option Conditional UBs to yes, we can obtain conditional upper bounds (shown in Figure 3.20).

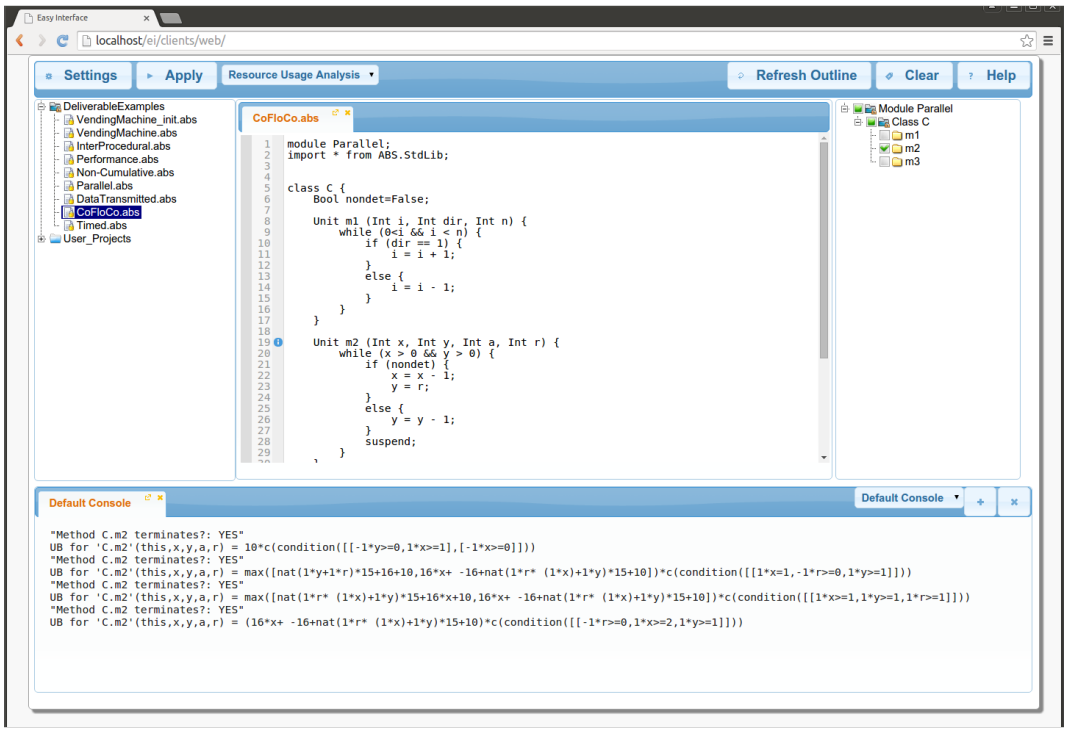


Figure 3.20: Conditional UBs for method C.m2 in the example of Figure 2.10 using CoFloCo

Chapter 4

Conclusions

This deliverable comprises the main achievements in the task of resource analysis during the first two years of the Envisage's project. The main goal has been to infer system level information about the resource consumption and to take deployment descriptions into account in the analysis. To achieve these goals, we have enhanced the may-happen-in-parallel analysis used in SACO. We have worked in the integration of scheduling policies in the analysis, in particular, we have considered priority-based scheduling, which is one of the policies more commonly adopted in practice. Another important contribution has been the extension of the may-happen-in-parallel analysis to handel inter-procedural synchronization. Our next goal has been to infer system-level resource analysis information. The first step is to be able to bound the resource consumption of tasks which interleave their execution. This is challenging because at the points in which tasks might interleave their execution, we must find out the tasks that can be executing and prove termination properties on them. Next, we infer the peak of the resource consumption at each location of a distributed system. This information is useful to dimensioning the distributed system: it will allow us to determine the size of each location *task queue*; the required size of the location's memory; and the processor execution speed required to guarantee a certain response time under heavy load (i.e. at the peak). We have also investigated the notion of parallel cost that allows us to infer an overall estimation of the resource consumption by exploiting the fact that computations across different distributed components are executed in parallel. We have investigated the inference of new performance indicators, like the load-balance, the level of parallelism achieved and the data transmission cost. The latter is an important contribution to be able to infer the response times of distributed components. In particular, if one knows the bandwidth conditions among each pair of locations, we can infer the time required to transmit the data and to retrieve the result. This time should be added to the time required to carry out the computation at each location, which is an orthogonal issue. Conversely, we can use our analysis to establish the bandwidth conditions required to ensure a certain response time. The resource analysis in time can be useful to investigate the resource consumption of a system according to various assumptions about the execution environment that take the form of time annotations. For example, these time annotations could be the result of previous resource analysis of the bandwidth usage. Finally, the techniques developed in CoFloCo widen the range of systems that can be analyzed and increase the precision of all the related analyses.

Bibliography

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193, 2007.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM, 2010.

[3] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer-Verlag, 2014.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. Cost Analysis of Concurrent OO programs. In Hongseok Yang, editor, *Proceedings 9th Asian Symposium on Programming Languages and Systems (APLAS 2011)*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, 2011.

[5] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[6] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[7] Elvira Albert, Jesús Correas, Einar Broch Johnsen, and Guillermo Román-Díez. Parallel Cost Analysis of Distributed Systems. In *Static Analysis - 22nd International Symposium, SAS 2015. Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 275–292. Springer-Verlag, 2015.

[8] Elvira Albert, Jesús Correas, Enrique Martín-Martín, and Guillermo Román-Díez. Static Inference of Transmission Data Sizes in Distributed Systems. In Tiziana Margaria and Bernhard Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*, pages 104–119. Springer-Verlag, 2014.

[9] Elvira Albert, Jesús Correas, Germán Puebla, and Guillermo Román-Díez. Quantified Abstract Configurations of Distributed Systems. *Formal Aspects of Computing*, 27(4):665–699, 2015.

[10] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Peak Cost Analysis of Distributed Systems. In *21st International Static Analysis Symposium (SAS'14)*, volume 8723 of *Lecture Notes in Computer Science*, pages 18–33. Springer-Verlag, 2014.

[11] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, volume 9035 of *Lecture Notes in Computer Science*, pages 85–100. Springer-Verlag, 2015.

[12] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. Analysis of may-happen-in-parallel in concurrent objects. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, volume 7273 of *Lecture Notes in Computer Science*, pages 35–51. Springer-Verlag, 2012.

[13] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. Termination and cost analysis of loops with concurrent interleavings. In Dang Van Hung and Mizuhito Ogawa, editors, *11th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 349–364. Springer-Verlag, October 2013.

[14] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis - 22nd International Symposium, SAS 2015. Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 72–89. Springer-Verlag, 2015.

[15] Elvira Albert, Samir Genaim, and Enrique Martin-Martin. May-Happen-in-Parallel Analysis for Priority-based Scheduling. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 8312 of *Lecture Notes in Computer Science*, pages 18–34. Springer-Verlag, December 2013.

[16] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *18th International Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer-Verlag, 2005.

[17] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Principles of Programming Languages*. ACM Press, 1978.

[18] Antonio Flores Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *12th Asian Symposium on Programming Languages and Systems (APLAS'14)*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, November 2014.

[19] Antonio Flores-Montoya and Reiner Hähnle. Resource consumption of concurrent objects over time. Technical report, TUD, 2015. `https://www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Page_Content/Group_Members/Antonio_Flores-Montoya/ResourceAnalysisTime_TechReport.pdf`.

[20] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of POPL'13*, pages 185–197. ACM, 2003.

[21] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.

[22] Jonathan K. Lee and Jens Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Principles and Practice of Parallel Programming (PPoPP'10)*, pages 25–36. ACM Press, 2010.

[23] Lin Li and Clark Verbrugge. A practical MHP information analysis for concurrent java programs. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *17th International Workshop on Languages and Compilers for High Performance Computing*, Lecture Notes in Computer Science, pages 194–208. Springer-Verlag, 2004.

Glossary

ABS stands for Abstract Behavioral Specification.

Abstract queue configuration contains an over-approximation of the sets of tasks that may be simultaneously queued in a location task queue during any execution of the program.

Bandwidth is a measurement of bit-rate of data communication resources expressed in bits per second or multiples of it.

Concurrent interleaving is the ability to interleave the execution of several processes in a location, either using explicit language constructs or preemptively.

Cost center is a symbolic construct used by resource analysis in cost expressions to distribute the cost among the components of a system.

Distributed Flow Graph (DFG) captures the different flows of execution that the distributed system can perform

Location is an abstract representation of a processing unit in the ABS language, which may refer to a conceptual processor, an actual processor, or a core in a multiprocessor.

May-happen-in-parallel analysis is a static analysis that infers the pairs of instructions in a program that may execute in parallel.

Non-cumulative cost analysis is a static analysis to estimate the maximum of the resource consumption for non-cumulative resources that increase and decrease along the computation (i.e., resources can be acquired and released)

Parallel Cost Analysis is a static analysis that infers the parallel cost of the program, i.e. a new notion of cost that takes into account that some tasks execute in parallel across different distributed components and thus the total cost is the maximum among them.

Peak cost analysis is a resource analysis that infers the maximum amount of resources that a location might require along any execution of the program.

Peak resource consumption is the maximum amount of resources that a location might require along any execution of the program.

Performance indicators are metrics that allow us to assess the quality of a distributed system, such us whether the load in the system is well-balanced (i.e., all distributed nodes execute a similar number of steps), or how the communication cost between nodes is distributed.

Priority-based scheduling is a scheduling policy that, given priorities assigned to tasks, selects the next task to be executed based on such priorities.

Quantified queue configuration is a bound on the peak resource consumption of a location.

Ranking function is a function that bounds the number of iterations of a loop in a program.

Release point is a program point in a concurrent or distributed program in which the processor is released and therefore other pending tasks can execute.

Rely-guarantee is a style of reasoning used for compositional verification and analysis of thread-based concurrent systems.

Resource is a cost metrics to be measured when analyzing a program. Classical measures are the number of executed instructions, memory usage, or the number of calls to a given method.

Resource analysis is a static analysis that aims at bounding the consumption of some resource by inspecting a program without having to actually run it.

Scheduling policy in a concurrent program is the policy used for selecting the task to be executed from the queue of pending tasks.

Strongly connected component is a set of nodes that form a cycle in a graph.

Task queue is a list of tasks pending to be executed in a location.

Term size is a measurement of the size of a functional data structure that counts 1 for each basic value or data type constructor.

Termination analysis is a static analysis that aims at inferring that a program terminates.

Total resource consumption is the total amount of resources that a location may consume in any execution of the program.

Transmission data size is the size of the data transmitted between two locations in any execution.

Upper bound is an expression that bounds the resource consumption of a program depending on some parameters, usually input arguments or fields.

Appendix A

Article *May-Happen-in-Parallel Analysis for Priority-based Scheduling*, [15]

May-Happen-in-Parallel Analysis for Priority-based Scheduling

Authors' Version*

Elvira Albert, Samir Genaim, and Enrique Martin-Martin

Complutense University of Madrid, Spain

Abstract. A *may-happen-in-parallel* (MHP) analysis infers the sets of pairs of program points that may execute in parallel along a program's execution. This is an essential piece of information to detect data races, and also to infer more complex properties of concurrent programs, e.g., deadlock freeness, termination and resource consumption analyses can greatly benefit from the MHP relations to increase their accuracy. Previous MHP analyses have assumed a worst case scenario by adopting a simplistic (non-deterministic) task scheduler which can select any available task. While the results of the analysis for a non-deterministic scheduler are obviously sound, they can lead to an overly pessimistic result. We present an MHP analysis for an asynchronous language with *prioritized* tasks buffers. *Priority-based scheduling* is arguably the most common scheduling strategy adopted in the implementation of concurrent languages. The challenge is to be able to take task priorities into account at static analysis time in order to filter out unfeasible MHP pairs.

1 Introduction

In asynchronous programming, programmers divide computations into shorter tasks which may create additional tasks to be executed asynchronously. Each task is placed into a task-buffer which can execute in *parallel* with other task-buffers. The use of a *synchronization* mechanism enables that the execution of a task is synchronized with the completion of another task. Synchronization can be performed via shared-memory [9] or via future variables [13, 8]. Concurrent *interleavings* in a buffer can occur if, while a task is awaiting for the completion of another task, the processor is released such that another pending task can start to execute. This programming model captures the essence of the concurrency models in X10 [13], ABS [12], Erlang [1] and Scala [11], and it is the basis of

* Appeared in the *Proc. of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-19)*. Springer, Lecture Notes in Computer Science volume 8312, subline *Advanced Research in Computing and Software Science (ARCoSS)*, 2013, pp 18–34. The final publication is available at link.springer.com.

actor-like concurrency [2, 11]. The most common strategy to schedule tasks is undoubtedly *priority-based scheduling*. Each task has a priority level such that when the active task executing in the buffer releases the processor, a highest priority pending task is taken from its buffer and begins executing. Asynchronous programming with *prioritized tasks buffers* has been used to model real-world asynchronous software, e.g., Windows drivers, engines of modern web browsers, Linux's work queues, among others (see [9] and its references).

The higher level of abstraction that asynchronous programming provides, when compared to lower-level mechanisms like the use of multi-threading and locks, allows writing software which is more reliable and more amenable to be analyzed. In spite of this, proving error-freeness of these programs is still quite challenging. The difficulties are mostly related to: (1) *Tasks interleavings*, typically a programmer decomposes a task t into subtasks $t_1, \ldots, t_n$. Even if each of the sub-tasks would execute serially, it can happen that a task k unrelated to this computation interleaves its execution between t_i and t_{i+1}. If this task k changes the shared-memory, it can interfere with the computation in several ways, e.g., leading to non-termination, to an unbounded resource consumption, and to deadlocks. (2) *Buffers parallelism*, tasks executing across several task-buffers can run in parallel, this could lead to deadlocks and data races.

In this paper, we present a *may-happen-in-parallel* (MHP) analysis which identifies pairs of statements that can execute in parallel and in an interleaved way (see [13, 3]). MHP is a crucial analysis to later prove the properties mentioned above. It directly allows ensuring absence of data races. Besides, MHP pairs allow us to greatly improve the accuracy of deadlock analysis [16, 10] as it discards unfeasible deadlocks when the instructions involved in a possible deadlock cycle cannot happen in parallel. Also, it improves the accuracy of termination and cost analysis [5] since it allows discarding unfeasible interleavings. For instance, consider a loop like `while (l!=null) {x=b.m(l.data); await x?; l=l.next;}`, where `x=b.m(e)` posts an asynchronous task `m(e)` on buffer `b`, and the instruction `await x?` synchronizes with the completion of the asynchronous task by means of the future variable `x`. If the asynchronous task is not completed (`x` is not ready), the current task releases the processor and another task can take it. This loop terminates provided no instruction that increases the length of the list `l` *interleaves* or *executes in parallel* with the body of this loop.

Existing MHP analyses [13, 3] assume a worst case scenario by adopting a simplistic (non-deterministic) task scheduler which can select any available task. While the results of the analysis for a non-deterministic scheduler are obviously sound, they can lead to an overly pessimistic result and report false errors due to unfeasible schedulings in the task order selection. For instance, consider two buffers `b1` and `b2` and assume we are executing a task in `b1` with the following code "`x=b1.m1(e1); y=b1.m2(e2); await x?; b2.m3(e3);`". If the priority of the task executing `m1` is smaller than that of `m2`, then it is ensured that task `m2` and `m3` will not execute in parallel even if the synchronization via **await** is on the completion of `m1`. This is because at the **await** instruction, when the

processor is released, `m2` will be selected by the priority-based scheduler before `m1`. A non-deterministic scheduler would give this spurious parallelism.

Our starting point is the MHP analysis for non-deterministic scheduling of [3], which distinguishes a local phase in which one inspects the code of each task locally, and ignores transitive calls, and a global phase in which the results of the local analysis are composed to build a global *MHP-graph* which captures the parallelism with transitive calls and among multiple task-buffers. The contribution of this paper is an MHP analysis for a priority-based scheduling which takes priorities into account both at the local and global levels of the analysis. As each buffer has its own scheduler which is independent of other buffer's schedulers, priorities can be only applied to establish the order of execution among the tasks executing on the same task-buffer (*intra-buffer* MHP pairs). Interestingly, even by only using priorities at the intra-buffer level, we are also able to implicitly eliminate unfeasible *inter-buffer* MHP pairs. We have implemented our analysis in the MayPar system [4] and evaluated it on some challenging examples, including some of the benchmarks used in [9]. The system can be used online through a web interface where the benchmarks used are also available.

2 Language

We consider asynchronous programs with priority-levels and multiple tasks buffers. Tasks can be synchronized with the completion of other tasks (of the same or of a different buffer) using futures. In this model, only highest-priority tasks may be dispatched, and tasks from different task buffers execute in parallel. The number of task buffers does not have to be known a priori and task buffers can be dynamically created. We keep the concept of task-buffer disconnected from physical entities, such as processes, threads, objects, processors, cores, etc. In [9], particular mappings of task-buffers to such entities in real-world asynchronous systems are described. Our model captures the essence of the concurrency and distribution models used in X10 [13] and in actor-languages (including ABS [12], Erlang [1] and Scala [11]). It also has many similarities with [9], the main difference being that the synchronization mechanism is by means of future variables (instead of using the shared-memory for this purpose).

2.1 Syntax

Each program declares a sequence of global variables $g_0, \ldots, g_n$ and a sequence of methods named $m_0, \ldots, m_i$ (that may declare local variables) such that one of the methods, named main, corresponds to the initial method which is never posted or called and it is executing in a buffer with identifier `0`. The grammar below describes the syntax of our programs. Here, T are types, m procedure names, e expressions, x can be global or local variables, buffer identifiers b are local variables, f are future variables, and priority levels p are natural numbers.

$$
\begin{array}{lll}
M & ::= & T\ m(\bar{T}\ \bar{x})\{s; \textsf{return}\ e;\} \\
s & ::= & s; s \mid x = e \mid \textsf{if}\ e\ \textsf{then}\ s\ \textsf{else}\ s \mid \textsf{while}\ e\ \textsf{do}\ s \mid \\
 & & \textsf{await}\ f? \mid b = \textsf{newBuffer} \mid f = b.m(\langle \bar{e} \rangle, p) \mid \textsf{release}
\end{array}
$$

The notation $\bar{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. We use the special buffer identifier *this* to denote the current buffer. For the sake of generality, the syntax of expressions is left free and also the set of types is not specified. We assume that every method ends with a **return** instruction.

The concurrency model is as follows. Each buffer has a lock that is shared by all tasks that belong to the buffer. Data synchronization is by means of future variables as follows. An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}b.m(\langle \bar{z} \rangle, p)$ such that **await** $y?$ is executed only when the future variable y is available (and hence the task executing m is finished). In the meantime, the buffer's lock can be released and some highest priority *pending* task on that buffer can take it. The instruction **release** can be used to unconditionally release the processor so that other pending task can take it. Therefore, our concurrency model is *cooperative* as processor release points are explicit in the code, in contrast to a *preemptive* model in which a higher priority task can interrupt the execution of a lower priority task at any point (see Sec. 7). W.l.o.g, we assume that all methods in a program have different names.

2.2 Semantics

A *program state* $St = \langle g, \texttt{Buf} \rangle$ is a mapping g from the global variables to their values along with all created buffers `Buf`. `Buf` is of the form $\mathit{buffer}_1 \parallel \ldots \parallel \mathit{buffer}_n$ denoting the parallel execution of the created task-buffers. Each *buffer* is a term $\mathit{buffer}(\mathit{bid}, \mathit{lk}, \mathcal{Q})$ where *bid* is the buffer identifier, lk is the identifier of the *active task* that holds the buffer's lock or $\perp$ if the buffer's lock is free, and $\mathcal{Q}$ is the set of tasks in the buffer. Only one task can be *active* (running) in each buffer and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $\mathit{tsk}(\mathit{tid}, m, p, l, s)$ where *tid* is a unique task identifier, m is the method name executing in the task, p is the task priority level (the larger the number, the higher the priority), l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created buffers and tasks never disappear from the state.

The execution of a program starts from an initial state where we have an initial buffer with identifier 0 executing task 0 of the form $S_0 = \langle g, \mathit{buffer}(0, 0, \{\mathit{tsk}(0, \textsf{main}, p, l, \mathit{body}(\textsf{main}))\}) \rangle$. Here, g contains initial values for the global variables, l maps parameters to their initial values and local reference and future variables to **null** (standard initialization), p is the priority given to main, and $\mathit{body}(m)$ refers to the sequence of instructions in the method m. The execution proceeds from S_0 by selecting *non-deterministically* one of the buffers and applying the semantic rules depicted in Fig. 1. We omit the treatment of the sequential instructions as it is standard, and we also omit the global memory g from the state as it is only modified by the sequential instructions.

$$(\textsc{newbuffer})\ \frac{\text{fresh}(bid')\ ,\ l' = l[x \to bid'],\ t = tsk(tid, m, p, l, \langle x = \mathsf{newBuffer}; s\rangle)}{\begin{array}{c}buffer(bid, tid, \{t\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, tid, \{tsk(tid, m, p, l', s)\} \cup \mathcal{Q}) \parallel buffer(bid', \bot, \{\}) \parallel B\end{array}}$$

$$(\textsc{priority})\ \frac{highestP(\mathcal{Q}) = tid,\ t = tsk(tid, \_, \_, \_, s) \in \mathcal{Q},\ s \neq \epsilon(v)}{buffer(bid, \bot, \mathcal{Q}) \parallel B \rightsquigarrow buffer(bid, tid, \mathcal{Q}) \parallel B}$$

$$(\textsc{async})\ \frac{\begin{array}{c}l(x) = bid_1,\ \text{fresh}(tid_1),\ l' = l[y \to tid_1],\ l_1 = buildLocals(\bar{z}, m_1) \\ buffer(bid, tid, \{tsk(tid, m, p, l, \langle y = x.m_1(\bar{z}, p_1); s\rangle\} \cup \mathcal{Q}) \parallel buffer(bid_1, \_, \mathcal{Q}') \parallel B \rightsquigarrow\end{array}}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l', s)\} \cup \mathcal{Q}) \parallel \\ buffer(bid_1, \_, \{tsk(tid_1, m_1, p_1, l_1, body(m_1))\} \cup \mathcal{Q}') \parallel B\end{array}}$$

$$(\textsc{await1})\ \frac{l(y) = tid_1,\ tsk(tid_1, \_, \_, \_, s_1) \in \mathtt{Buf}, s_1 = \epsilon(v)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, tid, \{tsk(tid, m, p, l, s)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$(\textsc{await2})\ \frac{l(y) = tid_1,\ tsk(tid_1, \_, \_, \_, s_1) \in \mathtt{Buf}, s_1 \neq \epsilon(v)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, \bot, \{tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$(\textsc{release})\ \frac{}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{release}; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, \bot, \{tsk(tid, m, p, l, s)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$(\textsc{return})\ \frac{v = l(x)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{return}\ x; \rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, \bot, \{tsk(tid, m, p, l, \epsilon(v))\} \cup \mathcal{Q}) \parallel B\end{array}}$$

Fig. 1. Summarized Semantics for a Priority-based Scheduling Async Language

Newbuffer: an active task *tid* in buffer *bid* creates a buffer bid' which is introduced to the state with a free lock. Priority: Function *highestP* returns a highest-priority task that is not finished, and it obtains its buffer's lock. Async: A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is associated to the corresponding future variable y in l'. We have assumed that $bid \neq bid_1$, but the case $bid = bid_1$ is analogous, the new task tid_1 is simply added to $\mathcal{Q}$ of *bid*. Await1: If the future variable we are awaiting for points to a finished task, the **await** can be completed. The finished task t_1 is looked up in all buffers in the current state (denoted `Buf`). Await2: Otherwise, the task yields the lock so that any other task of the same buffer can take it. Release: the current task frees the lock. Return: When **return**

```
1 // g1 global variable
2 // g2 global variable
3 void task(){
4     g2 = g2 + 1;
5 }
6 void f(){
7     while( g1 > 0 ){
8       g1 = g1 − 1;
9       g2 = g2 + 1;
10      release;
11    }
12 }
13 void m(){
14    while( g1 < 0 ){
15      g1 = g1 + 1;
16      release;
17    }
18 }
19 void h(){
20    while(g1 > 0){
21      g1 = g1 − 2;
22      release;
23    }
24 }
25 // main has priority 0
26 main(){
27    this.f(<>,10);
28    Fut x = this.m(<>,5);
29    await x?;
30    this.h(<>,10);
31    Buffer o=newbuffer;
32    o.task(<>,0);
33    ...
34 }
```

Fig. 2. Example for inter-buffer and intra-buffer may-happen-in-parallel relations

is executed, the return value is stored in v so that it can be obtained by the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$) but it does not disappear from the state as its return value may be needed later on in an **await**.

Example 1. Figure 2 shows some simple methods which will illustrate different aspects of our analysis. In particular, non-termination of certain tasks and data races can occur if priorities are not properly assigned by the programmer, and later considered by the analysis. Our analysis will take the assigned priorities into account in order to gather the necessary MHP information to be able to guarantee termination and absence of data races. Let us by now only show some execution steps. The execution starts from a buffer 0 with a single task in which we are executing the main method. Let us assume that such task has been given the lowest priority 0. The global memory g is assumed to be properly initialized.

$St_0 \equiv \langle g, \mathit{buffer}(0, 0, \{\mathit{tsk}(0, \mathsf{main}, 0, l, \mathit{body}(\mathsf{main}))\})\rangle \xrightarrow{\mathit{async}}$
$St_1 \equiv \langle g, \mathit{buffer}(0, 0, \{\mathit{tsk}(0, ..), \mathit{tsk}(1, f, 10, ..)\})\rangle \xrightarrow{\mathit{async}}$
$St_2 \equiv \langle g, \mathit{buffer}(0, 0, \{\mathit{tsk}(0, ..), \mathit{tsk}(1, ..), \mathit{tsk}(2, m, 5..)\})\rangle \xrightarrow{\mathit{await}}$
$St_3 \equiv \langle g, \mathit{buffer}(0, \bot, \{\mathit{tsk}(0, .., \mathit{await}), \mathit{tsk}(1, ..), \mathit{tsk}(2, m, 5..)\})\rangle \xrightarrow{\mathit{priority}}$
$St_4 \equiv \langle g, \mathit{buffer}(0, 1, \{\mathit{tsk}(0, .., \mathit{await}), \mathit{tsk}(1, ..), \mathit{tsk}(2, m, 5..)\})\rangle \rightarrow^*$
$St_5 \equiv \langle g', \mathit{buffer}(0, 1, \{\mathit{tsk}(0, .., \mathit{await}), \mathit{tsk}(1, .., \mathit{return}), \mathit{tsk}(2, m, 5..)\})\rangle \xrightarrow{\mathit{return}}$
$St_6 \equiv \langle g', \mathit{buffer}(0, \bot, \{\mathit{tsk}(0, .., \mathit{await}), \mathit{tsk}(1, .., \epsilon(v)), \mathit{tsk}(2, m, 5..)\})\rangle \xrightarrow{\mathit{priority}}$
$St_7 \equiv \langle g', \mathit{buffer}(0, 2, \{\mathit{tsk}(0, .., \mathit{await}), \mathit{tsk}(1, .., \epsilon(v)), \mathit{tsk}(2, m, 5..)\})\rangle \rightarrow^*$
$St_8 \equiv \langle g'', \mathit{buffer}(0, 0, \{\mathit{tsk}(0, ..), \mathit{tsk}(1, .., \epsilon(v)), \mathit{tsk}(2, .., \epsilon(v)), \mathit{tsk}(3..)\})\rangle \xrightarrow{\mathit{newbuf}}$
$St_9 \equiv \langle g'', \mathit{buffer}(0, 0, \{\mathit{tsk}(0..), \mathit{tsk}(1..), \mathit{tsk}(2..), \mathit{tsk}(3..)\}), \rangle \mathit{buffer}(1, \bot, \{\}) \xrightarrow{\mathit{async}}$
$St_{10} \equiv \langle g'', \mathit{buffer}(0, 0, \{\mathit{tsk}(0..), ..\}), \rangle \mathit{buffer}(1, \bot, \{\mathit{task}(4..)\}) \xrightarrow{\mathit{priority}}$
$St_{11} \equiv \langle g'', \mathit{buffer}(0, 0, \{\mathit{tsk}(0..), ..\}), \rangle \mathit{buffer}(1, 4, \{\mathit{task}(4..)\}) \xrightarrow{...}$

At St_1, we execute the instruction at Line 27 (L27 for short) that posts, in the current buffer **this**, a new task (with identifier 1) that will execute method `f` with priority 10. The next step St_2 posts another task (with identifier 2) in the current buffer with a lower priority (namely 5). At St_3, an **await** instruction (L29) is used to synchronize the execution with the completion of the task 2 spawned at L28. As the task executing `f` has higher priority than the one executing `m`, it will be selected for execution at St_4. After returning from the execution of task 1 in St_5, the PRIORITY rule selects task 2 for execution in St_6. An interesting aspect is that after creating buffer 1 at St_{10}, execution can non-deterministically choose buffer 0 or 1 (in St_{11} buffer 1 has been selected).

3 Definition of MHP

We first formally define the concrete property "MHP" that we want to approximate using static analysis. In what follows, we assume that instructions are labelled such that it is possible to obtain the corresponding program point identifiers. We also assume that program points are globally different. We use $p_{\hat{m}}$ to refer to the entry program point of method m, and $p_{\dot{m}}$ to all program points after its **return** instruction. The set of all program points of P is denoted by $P_{\mathcal{P}}$. We write $p \in m$ to indicate that program point p belongs to method m. Given a sequence of instructions s, we use $pp(s)$ to refer to the program point identifier associated with its first instruction and $pp(\epsilon(v)) = p_{\dot{m}}$.

Definition 1 (concrete MHP). *Given a program P, its MHP is defined as* $\mathcal{E}_P = \cup\{\mathcal{E}_S \mid S_0 \rightsquigarrow^* S\}$ *where for* $S = \langle g, \texttt{Buf}\rangle$, *the set* $\mathcal{E}_S$ *is* $\mathcal{E}_S = \{(pp(s_1), pp(s_2)) \mid buffer(bid_1, \_, \mathcal{Q}_1) \in \texttt{Buf}, buffer(bid_2, \_, \mathcal{Q}_2) \in \texttt{Buf}, t_1 = tsk(tid_1, \_, \_, \_, s_1) \in \mathcal{Q}_1, t_2 = tsk(tid_2, \_, \_, \_, s_2) \in \mathcal{Q}_2, tid_1 \neq tid_2\}$.

The above definition considers the union of the pairs obtained from all derivations from S_0. This is because execution is non-deterministic in two dimensions: (1) in the selection of the buffer that is chosen for execution, since the buffers have access to the global memory different behaviours (and thus MHP pairs) can be obtained depending on the execution order, and (2) when there is more than one task with the highest priority, the selection is non-deterministic.

The MHP pairs can originate from *direct* or *indirect* task creation relationships. For instance, the parallelism between the points of the tasks executing `h` and `task` is *indirect* because they do not invoke one to the other directly, but a third task `main` invokes both of them. However, the parallelism between the points of the task `main` and those of `task` is *direct* because the first one invokes directly the latter one. Def. 1 captures all these forms of parallelism.

Importantly, $\mathcal{E}_P$ includes both *intra-buffer* and *inter-buffer* MHP pairs, each of which are relevant for different kinds of applications, as we explain below.

Intra-buffer MHP pairs. Intra-buffer relations in Def. 1 are pairs in which $bid_1 \equiv bid_2$. We always have that the first instructions of all tasks which are pending in the buffer's queue may-happen-in-parallel among them, and also with the instruction of the task which is currently active (has the buffer's lock). This piece of information allows approximating the tasks interleavings that we may have in a considered buffer. In particular, when the execution is at a processor release point, we use the MHP pairs to see the instructions that may execute if the processor is released. Information about task interleavings is essential to infer termination and resource consumption in any concurrent setting (see [5]).

Example 2. Consider the execution trace in Ex. 1, we have the MHP pairs (29,$p_{\mathring{f}}$) and (29,$p_{\mathring{m}}$) since when the active task 0 is executing the **await** (point 29) in St_4, we have that tasks 1 and 2 are pending at their entry points. The following execution steps give rise to many other MHP pairs. The most relevant point to note is that in St_8 when the execution is at L30 and onwards, the tasks 1 and 2 are guaranteed to be at their exit program points $p_{\dot{f}}$ and $p_{\dot{m}}$. Thus, we will not have any MHP pair between the instructions that update the global variable `g1` (L8 and L15 in tasks 1 and 2, resp.) and the release point at L22 of the task 3 executing `h`. This information is essential to prove the termination of `h`, as the analysis needs to be sure that the loop counter cannot be modified by instructions of other tasks that may execute in parallel with the body of this loop. The information is also needed to obtain an upper bound on the number of iterations of the loop and then infer the resource consumption of `h`.

Inter-buffer MHP pairs. In addition to intra-buffer MHP relations, *inter-buffer* MHP pairs happen when $bid_1 \neq bid_2$. In this case, we obtain the instructions that may execute in parallel in different buffers. This information is relevant at least for two purposes: (1) to detect data-races in the access to the global memory and (2) to detect deadlocks and livelocks when one buffer is awaiting for the completion of one task running in another buffer, while such other task is awaiting for the completion of the current task, and the execution of these (synchronization) instructions happens in parallel (or simultaneously). If the language allows blocking the execution of the buffer such that no other pending task can take it, we have a deadlock, otherwise we have a livelock.

Example 3. Consider again the execution trace in Ex. 1, in St_{10} we have created a new buffer 1 in which task 4 starts to execute at St_{11}. We will have the inter-buffer pair (21,4) as we can have L21 executing in buffer 0 and L4 executing in buffer 1. Note that, if `task` had updated `g1` instead of updating `g2`, we would have had a data race. Data races can lead to different types of errors, and static analyses that detect them are of utmost importance.

4 Method-Level Analysis with Priorities

In this section, we present the local phase of our MHP analysis which assigns to each program point, of a given method, an abstract state that describes the

$$
\begin{array}{lrcll}
(1) & \tau_{\mathtt{p}}(y{=}\textsf{this}.m(\bar{x},p),M) & = & M[\langle y,O,Z,R\rangle/\langle\star,O,Z,R\rangle]\cup\{\langle y,\mathtt{t},\check{m},p\rangle\} & \\
(2) & \tau_{\mathtt{p}}(y{=}x.m(\bar{x},p),M) & = & M[\langle y,O,Z,R\rangle/\langle\star,O,Z,R\rangle]\cup\{\langle y,\mathtt{o},\tilde{m},p\rangle\} & \\
(3) & \tau_{\mathtt{p}}(\textsf{release},M) & = & \tau_{\mathtt{p}}(\textsf{release}_1;\textsf{release}_2,M) & \\
(4) & \tau_{\mathtt{p}}(\textsf{release}_1,M) & = & M[\langle Y,\mathtt{t},\check{m},p\rangle/\langle Y,\mathtt{t},\tilde{m},p\rangle] & \text{where } p\geq\mathtt{p} \\
(5) & \tau_{\mathtt{p}}(\textsf{release}_2,M) & = & M[\langle Y,\mathtt{t},\tilde{m},p\rangle/\langle Y,\mathtt{t},\hat{m},p\rangle] & \text{where } p>\mathtt{p} \\
(6) & \tau_{\mathtt{p}}(\textsf{await } y?,M) & = & M'[\langle y,O,\tilde{m},R\rangle/\langle y,O,\hat{m},R\rangle] & \\
 & & & \text{where } M'=\tau_{\mathtt{p}}(\textsf{release}_1;\textsf{release}_2,M) & \\
(7) & \tau_{\mathtt{p}}(\textsf{return},M) & = & M[\langle Y,\mathtt{t},\check{m},R\rangle/\langle Y,\mathtt{t},\tilde{m},R\rangle] & \\
(8) & \tau_{\mathtt{p}}(b,M) & = & M \quad \text{otherwise} &
\end{array}
$$

Fig. 3. Method-level MHP transfer function: $\tau_{\mathtt{p}} : s \times \mathcal{B} \mapsto \mathcal{B}$.

status of the tasks that have been locally invoked so far. The status of a task can be (1) *pending*, i.e., it is at the entry program point; (2) *finished*, i.e., it has executed a **return** instruction already; or (3) *active*, i.e., it can be executing at any program point (including the entry and the exit). The analysis uses *MHP atoms* which are syntactic objects of the form $\langle F, O, T, R\rangle$ where

- F is either a valid future variable name or $\star$. The value $\star$ indicates that the task might not be associated with any future variable, either because there is no need to synchronize with its result, or because the future has been reused and thus the association lost (this does not happen in our example);
- O is the *buffer name* that can be $\mathtt{t}$ or $\mathtt{o}$, which resp. indicate that the task is executing on the same buffer or *maybe* on a different one;
- T can be $\check{m}$, $\tilde{m}$, or $\hat{m}$ where m is a method name. It indicates that the corresponding task is an instance of method m, and its status can be *pending*, *active*, or *finished* resp.;
- P is a natural number indicating the priority of the corresponding task.

Intuitively, an MHP atom $\langle F, O, T, R\rangle$ is read as follows: task T might be executing (in some status) on buffer O with priority P, and one can wait for it to finish using future variable F. The set of all MHP atoms is denoted by $\mathcal{A}$.

Example 4. The MHP atom $\langle x, \mathtt{t}, \tilde{m}, 5\rangle$ indicates that there is an instance of method m running in parallel, in the same buffer. This task is active (i.e., can be at any program point), has priority 5, and is associated with the future x. The MHP atom $\langle\star, \mathtt{o}, \hat{task}, 0\rangle$ indicates that there is an instance of method *task* running in parallel, maybe in a different buffer. This task is finished (i.e., has executed **return**), has priority 0, and it is associated to any future variable.

An abstract state is a multiset of MHP atoms from $\mathcal{A}$. The set of all multisets over $\mathcal{A}$ is denoted by $\mathcal{B}$. Given $M \in \mathcal{B}$, we write $(a, i) \in M$ to indicate that a appears exactly $i > 0$ times in M. We omit i when it is 1. The local analysis is applied on each method and, as a result, it assigns an abstract state from $\mathcal{B}$ to each program point in the program. The analysis takes into account the priority of the method being analyzed. Thus, since a method might be called with different priorities $\mathtt{p}_1, \ldots, \mathtt{p}_n$, the analysis should be repeated for each $\mathtt{p}_i$. For

the sake of simplifying the presentation, we assume that each method is always called with the same priority. Handling several priorities is a context-sensitive analysis problem that can be done by, e.g., cloning the corresponding code.

The analysis of a given method, with respect to priority $\mathtt{p}$, abstractly executes its code over abstract elements from $\mathcal{B}$. This execution uses a transfer function $\tau_{\mathtt{p}}$, depicted in Fig. 3, to rewrite abstract states. Given an instruction b and an abstract state $M \in \mathcal{B}$, $\tau_{\mathtt{p}}(b, M)$ computes a new abstract state that results from abstractly executing b in state M. Note that the subscript $\mathtt{p}$ in $\tau_{\mathtt{p}}$ is the priority of the method being analyzed. Let us explain the different cases of $\tau_{\mathtt{p}}$:

- (1) Posting a task on the same buffer adds a new MHP atom $\langle y, \mathtt{t}, \check{m}, p\rangle$ to the abstract state. It indicates that an instance of m is pending, with priority p, on the *same* buffer as the analyzed method, and is associated with future variable y. In addition, since y is assigned a new value, those atoms in M that were associated with y should now be associated with $\star$ in the new state. This is done by $M[\langle y, O, Z, R\rangle/\langle\star, O, Z, R\rangle]$ which replaces each atom that matches $\langle y, O, Z, R\rangle$ in M by $\langle\star, O, Z, R\rangle$;
- (2) It is similar to (1), the difference is that the new task might be posted on a buffer different from that of the method being analyzed. Thus, its status should be *active* since, unlike (1), it might start to execute immediately;
- (3)-(5) These cases highlight the use of priorities, and thus mark the main differences wrt [3]. They state that when releasing the processor, only tasks of equal or higher priorities are allowed to become active (simulated through $\mathsf{release}_1$). Moreover, when taking the control back, any task with strictly higher priority is guaranteed to have been finished (simulated through $\mathsf{release}_2$). Importantly, the abstract element after $\mathsf{release}_1$ is associated to the program point of the **release** instruction, and that after $\mathsf{release}_2$ is associated to the program point after the **release** instruction. These two auxiliary instructions are introduced to simulate the implicit "loop" (in the semantics) when the task is waiting at that point;
- (6) This instruction is similar to **release**, the only difference is that the status of the tasks that are associated with future variable y become finished in the following program point. Importantly, the abstract element after $\mathsf{release}_1$ is associated to the program point of the **await** y?;
- (7) It changes the status of every pending task executing on the same buffer to active, this is because the processor is released. Note that we do not consider priorities in this case, since the task is finished.

In addition to using the transfer function for abstractly executing basic instructions, the analysis merges the results of paths (in conditions, loops, etc) using a join operator. We refer to [3] for formal definitions of the basic abstract interpretations operators. In what follows, we assume that the result of the local phase is given by means of a mapping $\mathcal{L}_{P}{:}P_{\mathcal{P}} \mapsto \mathcal{B}$ which maps each program point p (including entry and exit points) to an abstract state $\mathcal{L}_{P}(p) \in \mathcal{B}$.

Example 5. Applying the local analysis on main, results in the following abstract states (initially the abstract state is $\emptyset$):

$28{:}\{\langle\star, \mathtt{t}, \check{\mathtt{f}}, 10\rangle\}$
$29{:}\{\langle\star, \mathtt{t}, \tilde{\mathtt{f}}, 10\rangle, \langle\mathtt{x}, \mathtt{t}, \tilde{\mathtt{m}}, 5\rangle\}$
$30{:}\{\langle\star, \mathtt{t}, \hat{\mathtt{f}}, 10\rangle, \langle\mathtt{x}, \mathtt{t}, \hat{\mathtt{m}}, 5\rangle\}$
$31{:}\{\langle\star, \mathtt{t}, \hat{\mathtt{f}}, 10\rangle, \langle\mathtt{x}, \mathtt{t}, \hat{\mathtt{m}}, 5\rangle, \langle\star, \mathtt{t}, \check{\mathtt{h}}, 10\rangle\}$
$32{:}\{\langle\star, \mathtt{t}, \hat{\mathtt{f}}, 10\rangle, \langle\mathtt{x}, \mathtt{t}, \hat{\mathtt{m}}, 5\rangle, \langle\star, \mathtt{t}, \check{\mathtt{h}}, 10\rangle\}$
$33{:}\{\langle\star, \mathtt{t}, \hat{\mathtt{f}}, 10\rangle, \langle\mathtt{x}, \mathtt{t}, \hat{\mathtt{m}}, 5\rangle, \langle\star, \mathtt{t}, \check{\mathtt{h}}, 10\rangle, \langle\star, \mathtt{o}, \tilde{\mathtt{task}}, 0\rangle\}$

Note that in the abstract state at program point 30 we have both f and m finished, this is because they have higher priority than main, and thus, while main is waiting at program point 29 both f and m must have completed their execution before main can proceed to the next instruction. If we ignore priorities, then we would infer that f might be active at program point 30 (which is less precise).

5 MHP Graph for Priority-based Scheduling

In this section we will construct a MHP graph relating program points and methods in the program, that will be used to extract precise information on which program points might globally run in parallel. In order to build this graph, we use the local information computed in Sec. 4 which already takes priorities into account. In Sec. 5.2, we explain how to use the MHP graph to infer the MHP pairs in the program. Finally, in Sec. 5.3 we compare the inference method of MHP pairs using a priority-based scheduling with the technique introduced in [3] for programs with a non-deterministic scheduling.

5.1 Construction of the MHP Graph with Priorities

The MHP graph has different types of nodes and different types of edges. There are nodes that represent the status of methods (active, pending or finished) and nodes that represent the program points. Outgoing edges from method nodes are unweighted and unlabeled, they represent points of which at most one might be executing. Outgoing edges from program point nodes are *labeled*, written $\rightarrow_l$ where the label l is a tuple (O, R) that contains a priority R and a buffer name O. These edges represent tasks such that any of them might be running. Besides, when two nodes are directly connected by $i > 1$ edges, we connect them with a single edge superscripted with *weight* i, written as $\rightarrow_l^i$ where l is the label as before.

Definition 2 (MHP graph with priorities). *Given a program P, and its method-level MHP analysis result $\mathcal{L}_P$, the MHP graph of P is a directed graph $\mathcal{G}_P = \langle V, E\rangle$ with a set of nodes V and a set of edges $E = E_1 \cup E_2$ defined:*

$$
\begin{aligned}
V &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \\
E_1 &= \{\tilde{m} \rightarrow p \mid m \in P_{\mathcal{M}}, p \in P_{\mathcal{P}}, p \in m\} \cup \{\hat{m} \rightarrow p_{\check{m}}, \check{m} \rightarrow p_{\check{m}} \mid m \in P_{\mathcal{M}}\} \\
E_2 &= \{p \rightarrow^i_{(O,R)} x \mid p \in P_{\mathcal{P}}, (\langle \_, O, x, R\rangle, i) \in \mathcal{L}_P(p)\}
\end{aligned}
$$

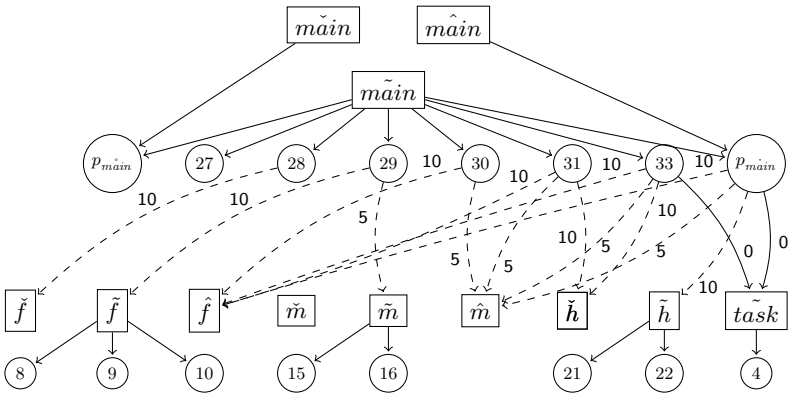


Fig. 4. MHP graph with priorities of the example

Example 6. Fig. 4 depicts the relevant fragment of the MHP graph for our running example. The graph only shows selected program points, namely all points of the main task and those points of the other tasks in which there is a **release** instruction, or in which the global memory is updated. For each task, we have three nodes which correspond to their possible status (except for h and task that we have omitted status that do not have incoming edges). In order to avoid cluttering the graph, in edges from program points, the labels only show the priority. The weight is omitted as it is always 1. The label corresponding to the buffer name is depicted using different types of arrows: normal arrows correspond to the buffer name o, while dashed arrows to t. From the pending (resp. finished) nodes, we always have an edge to the task entry (resp. exit) point. From the active nodes, we have edges to all program points in the corresponding method body, meaning that only one of them can be executing. The key aspect of the MHP graph is how we integrate the information gathered by the local analysis (with priorities) to build the edges from the program points: we can observe that node 28 has an edge to pending f, and at the **await** (node 29) the edges go to active f and m. After **await**, in nodes 30 and the next ones, the edges go to finished tasks. The remaining tasks only have edges to their program points since they do not make calls to other tasks.

5.2 Inference of Priority-based MHP pairs

The inference of MHP pairs is based on the notion of *intra-buffer path* in the MHP graph. A path from p_1 to p_2 is called *intra-buffer* if the program points p_1 and p_2 are reachable only through tasks in the same buffer. A simple way to ensure the intra-buffer condition is by checking that the buffer labels are always of type t (more accurate alternatives are discussed later). Intuitively, two program points $p_1, p_2 \in P_{\mathcal{P}}$ may run in parallel if one of the following conditions hold:

1. there is a non-empty path in $\mathcal{G}_{\mathcal{P}}$ from p_1 to p_2 or vice-versa; or
2. there is a program point $p_3 \in P_{\mathcal{P}}$, and non-empty *intra-buffer* paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$, and the *minimum priority* in both paths is the same; or
3. there is a program point $p_3 \in P_{\mathcal{P}}$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$, and at least one of the paths is not intra-buffer.

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task running from which it is possible to *transitively* reach p_2, or vice-versa. For instance (33,4) is a direct MHP resulting from the direct call from main to task.

The second and third cases correspond to *indirect MHP* scenarios in which a task is running at p_3 and there are two other tasks p_1 and p_2 executing in parallel and both are reachable from p_3. However, the second condition takes advantage of the priority information in intra-buffer paths to discard potential MHP pairs: if the minimum priority of path $pt_1 \equiv p_3 \rightsquigarrow p_1$ is lower than the minimum priority of $pt_2 \equiv p_3 \rightsquigarrow p_2$, then we are sure that the task containing the program point p_2 will be finished before the task containing p_1 starts. For instance, consider the two paths from 29 to 8 and from 29 to 16, which form the potential MHP pair (8,16). They are both intra-buffer (executing on buffer 0) and the minimum priority is not the same (the one to 16 has lower priority). Thus, (16,8) is not an MHP pair. The intuition is that the task with minimum priority (m in this case) will be *pending* and will not start its execution until all the tasks in the other path are finished. Similarly, we obtain that the potential MHP pair (10,15) is not a real MHP pair. Knowing that (10,15) and (16,8) are not MHP pairs is important because this allows us to prove termination of both tasks executing m and f. This is an improvement over the standard MHP analysis in [3], where they are considered as MHP pairs—see Sect. 5.3. On the other hand, when a path involves tasks running in several buffers (condition 3), priorities cannot be taken into account, as the buffers (and their task schedulers) work independently. Observe that, in the second and third conditions, the first edge can only be shared if it has weight $i > 1$ because it denotes that there might be more than one instance of the same type of task running. For instance, if we add the instruction o.task(<>,0) at L33 we will infer the pair (4,4), reporting a potential data race in the access to g2.

Let us formalize the inference of the priority-based MHP pairs. We write $p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$ to indicate that there is a path from p_1 to p_2 in $\mathcal{G}_{\mathcal{P}}$ such that the sum of the edges weights is greater than or equal to 1, and $p_1 \rightarrow^i x \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$ to mark that the path starts with an edge to x with weight i. We will say that a path $p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$ is *intra-buffer* if all the edges from program points to methods have t labels. Similarly, we will say that p is the *lowest priority of the path* $p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$, written $lowestP(p_1 \rightsquigarrow p_2) = p$, if p is the smallest priority

of all those that appear in edges from program points to methods in the path. We now define the *priority-based MHP pairs* as follows.

Definition 3. *Given a program P, we let $\tilde{\mathcal{E}}_P = D \cup I_{intra} \cup I_{inter}$ where*

$$
\begin{aligned}
D = \{(p_1,p_2) \mid\ & p_1,p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P)\}\\
I_{intra} = \{(p_1,p_2) \mid\ & p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P,\\
& p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \ \textit{is intra}-\textit{buffer}, \textit{lowestP}(p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1) = pr_1,\\
& p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \ \textit{is intra}-\textit{buffer}, \textit{lowestP}(p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2) = pr_2,\\
& (x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)) \wedge pr_1 = pr_2\}\\
I_{inter} = \{(p_1,p_2) \mid\ & p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P,\\
& p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \ \textit{or}\ p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \ \textit{are not intra}-\textit{buffer},\\
& x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\}
\end{aligned}
$$

An interesting point is that even if priorities can only be taken into account at an intra-buffer level, due to the inter-buffer synchronization operations, they allow discarding unfeasible MHP pairs at an inter-buffer level. For instance, we can see that (4,9), which would report an spurious data race, is not an MHP pair. Note that 4 and 9 execute in different buffers. Still, the priority-based local analysis has allowed us to infer that after 29, task `f` will be finished and thus, it cannot happen in parallel with the execution of `task` in buffer `o`. Thus, it is ensured that there will not be a data-race in the access to `g2` from the two different buffers.

The following theorem states the soundness of the analysis, namely, that $\tilde{\mathcal{E}}_P$ is an over-approximation of $\mathcal{E}_P$—the proof appears in the extended version of this paper [6]. Let $\mathcal{E}_P^{non-det}$ be the MHP pairs obtained by [3].

Theorem 1 (soundness). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P \subseteq \mathcal{E}_P^{non-det}$.

As we have discussed above, a sufficient condition for ensuring the intra-buffer condition of paths is to take priorities into account when all edges are labelled with the `t` buffer. However, if buffers can be uniquely identified at analysis time (as in the language of [9]), we can be more accurate. In particular, instead of using `o` to refer to any buffer, we would use the proper buffer name in the labels of the edges. Then, the intra-buffer condition will be ensured by checking that the buffer name along the considered paths is always the same.

In our language, buffers can be dynamically created, i.e., the number of buffers is not fixed a priori and one could have even an unbounded number of buffers (e.g., using `newBuffer` inside a loop). The standard way to handle this situation in static analysis is by incorporating points-to information [17, 15] which allows us to over-approximate the buffers created. A well-known approximation is by buffer creation site such that all buffers created at the same program point are abstracted by a single abstract name. In this setting, we can take advantage of the priorities (and apply case 2 in Def. 3) only if we are sure that an abstract name is referring to a single concrete buffer. As the task scheduler of each buffer works independently, we cannot use knowledge on the priorities to discard pairs if the abstract buffer might correspond to several concrete buffers. The extension of our framework to handle these cases is subject if future work.

5.3 Comparison with non-Priority MHP Graphs

The new MHP graphs with priority information (Sec. 5.1), and the conditions to infer MHP pairs (Sec. 5.2), are extensions of the corresponding notions in [3]. The original MHP graphs were defined as in Def. 2 with the following differences:

- The edges in E_2 do not contain the label *(O,R)* with the buffer name and the priority, but only the weight.
- The method-level analysis $\mathcal{L}_P(p)$ in [3] does not take priorities into account, so after a **release** instruction, pending tasks are set to active. With the method-level analysis in this paper (Sect. 4), tasks with a higher priority in the same buffer are set to finished after a **release** instruction—case (4) in Fig. 3. This generates less paths in the resulting MHP graph with priorities and therefore less MHP pairs.
- In [3], there is another type of nodes (future variable nodes) used to increase the accuracy when the same future variable is re-used in several calls in branching instructions. For the sake of simplicity we have not included future nodes here as their treatment would be identical as in [3].

Regarding the conditions to infer MHP pairs, only two are considered in [3]:

1. there is a non-empty path in $\mathcal{G}_P$ from p_1 to p_2 or vice-versa; or
2. there is a program point $p_3 \in P_\mathcal{P}$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$.

The first case is the same as the first condition in Sect 5.2. The second case corresponds to indirect MHP scenarios and is a generalization of conditions 2 and 3 in Sect 5.2 without considering priorities and intra-buffer paths. With these conditions, we have that the **release** point 22 cannot happen in parallel with the instructions that modify the value of the loop counter `g1` (namely 8 and 15), because there is no direct or indirect path connecting them starting from a program point. However, we have the indirect MHP pairs (10,15) and (16,8), meaning respectively that at the release point of `f` the counter `g1` can be modified by an interleaved execution of `m` and that at the release point of `m` the counter `g1` can be modified by an interleaved execution of `f`. Such spurious interleavings prevent us from proving termination of the tasks executing `f` and `m` and, as we have seen in Sec. 5.2, they are eliminated with the new MHP graphs with priorities and the new conditions for inferring MHP pairs.

6 Implementation in the MayPar System

We have implemented our analysis in a tool called MayPar [4], which takes as input a program written in the ABS language [12] extended with priority annotations. ABS is based on the concurrency model in Sec. 2 and uses the concept of *concurrent object* to realize the concept of task-buffer, such that object creation corresponds to buffer creation, and a method call `o.m()` posts

a task executing m on the queue of object o. Currently the annotations are provided at the level of methods, instead of at the level of tasks. This is because we lacked the syntax in the ABS language to include annotations in the calls, but the adaptation to calls will be straightforward once we have the parser extended.

We have made our implementation and a series of examples available online at `http://costa.ls.fi.upm.es/costabs/mhp`. After selecting an example, the analysis options allow: the selection of the entry method, enabling the option to consider priorities in the analysis, and several other options related to the format for displaying the analysis results and the verbosity level. After the analysis, MayPar yields in the output the MHP pairs in textual format and also optionally a graphical representation of the MHP graph. Besides, MayPar can be used in an interactive way which allows the user to select a line and the tool highlights all program points that may happen in parallel with it.

The examples on the MayPar site that include priority annotations are within the folder priorities. It is also possible to upload new examples by writing them in the text area. In order to evaluate our proposal, we have included a series of small examples that contain challenging patterns for priority-based MHP analysis (including our running example) and we have also encoded the examples in the second experiment of [9] and adapted them to our language (namely we use **await** on futures instead of assume on heap values). MayPar with priority-scheduling can successfully analyze all of them. Although these examples are rather small programs, this is not due to scalability limits of MayPar. It is rather because of the modeling overhead required to set up actual programs for static analysis.

7 Conclusions and Related Work

May-happen-in-parallel relations are of utmost importance to guarantee the sound behaviour of concurrent and parallel programs. They are a basic component of other analyses that prove termination, resource consumption boundness, data-race and deadlock freeness. As our main contribution, we have leveraged an existing MHP analysis developed for a simplistic scenario in which any task could be selected for execution in order to take task-priorities into account. Interestingly, have succeeded to take priorities into account both at the intra-buffer level and, indirectly, also at an inter-buffer level.

To the best of our knowledge, there is no previous MHP analysis for a priority-based scheduling. Our starting point is the MHP analysis for concurrent objects in [3]. Concurrent objects are almost identical to our multi-buffer asynchronous programs. The main difference is that, instead of buffers, the concurrency units are the objects. The language in [3] is data-race free because it is not allowed to access an object field from a different object. Our main novelty w.r.t. [3] is the integration of the priority-based scheduler in the framework. Although we have considered a cooperative concurrency model in which processor release points are explicit in the program, it is straightforward to handle a preemptive scheduling at the intra-buffer level like in [9], by simply adding a release point after posting a new task. If the posted task has higher priority, the active task will

be suspended and the posted task will become active. Thus, our analysis works directly for this model as well. As regards analyses for Java-like languages [14, 7], we have that a fundamental difference with our approach is that they do not take thread-priorities into account nor consider any synchronization between the threads as we do. To handle preemptive scheduling at the inter-buffer level, one needs to assume processor release points at any instruction in the program, and then the main ideas of our analysis would be applicable. However, we believe that the loss of precision could be significant in this setting.

Acknowledgements

This work was funded partially by EU project FP7-ICT-610582 ENVISAGE: *Engineering Virtualized Services* (`http://www.envisage-project.eu`), by the Spanish projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and by the Madrid Regional Government project S2009TIC-1465. We also want to acknowledge Antonio Flores-Montoya for his help and advice when implementing the analysis in the MayPar system.

References

1. Ericsson AB. *Erlang Efficiency Guide*, 5.8.5 edition, October 2011. From `http://www.erlang.org/doc/efficiency_guide/users_guide.html`.
2. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *FORTE'12*, *LNCS 7273*, pages 35–51. Springer, 2012.
4. E. Albert, A. Flores-Montoya, and S. Genaim. Maypar: a May-Happen-in-Parallel Analyzer for Concurrent Objects. In *SIGSOFT/FSE'12*, pages 1–4. ACM, 2012.
5. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*. To appear.
6. E. Albert, S. Genaim, and E. Martin-Martin. May-Happen-in-Parallel Analysis for Priority-based Scheduling (Extended Version). Technical Report SIC 12/13. Univ. Complutense de Madrid, 2013.
7. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *LCPC'05*, volume 4339 of *LNCS*, pages 152–169. Springer, 2005.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. M. Emmi, A. Lal, and S. Qadeer. Asynchronous programs with prioritized task-buffers. In *SIGSOFT FSE*, page 48. ACM, 2012.
10. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, Lecture Notes in Computer Science, pages 273–288. Springer, 2013.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
12. 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.

13. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Proc. of PPoPP'10*, pages 25–36. ACM, 2010.
14. L. Li and C. Verbrugge. A practical mhp information analysis for concurrent java programs. In *LCPC'04*, LNCS, pages 194–208. Springer, 2004.
15. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *ISSTA*, pages 1–11, 2002.
16. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of ICSE*, pages 386–396. IEEE, 2009.
17. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

Appendix B

Article *Termination and Cost Analysis of Loops with Concurrent Interleavings*, [13]

Termination and Cost Analysis of Loops with Concurrent Interleavings *

Elvira Albert[1], Antonio Flores-Montoya[2], Samir Genaim[1], and Enrique Martin-Martin[1]

[1] Complutense University of Madrid (UCM), Spain
[2] Technische Universität Darmstadt (TUD), Germany

Abstract. By following a *rely-guarantee* style of reasoning, we present a novel termination analysis for concurrent programs that, in order to prove termination of a considered loop, makes the assumption that the "shared-data that is involved in the termination proof of the loop is modified a finite number of times". In a subsequent step, it proves that this assumption holds in all code whose execution might interleave with such loop. At the core of the analysis, we use a *may-happen-in-parallel* analysis to restrict the set of program points whose execution can interleave with the considered loop. Interestingly, the same kind of reasoning can be applied to infer *upper bounds* on the number of iterations of loops with concurrent interleavings. To the best of our knowledge, this is the first method to automatically bound the cost of such kind of loops.

1 Introduction

We develop new techniques for cost and termination analyses of *concurrent objects*. The *actor*-based paradigm [1] on which concurrent objects are based has evolved as a powerful computational model for defining distributed and concurrent systems. In this paradigm, actors are the universal primitives of concurrent computation: in response to a message, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. *Concurrent objects* (a.k.a. active objects) [18,19] are actors which communicate via *asynchronous* method calls. Each concurrent object is a monitor and allows at most one *active* task to execute within the object. Scheduling among the tasks of an object is cooperative (or non-preemptive) such that a task has to release the object lock explicitly. Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and start to execute. The synchronization between the caller and the callee methods can be performed when the result is necessary by means of *future variables* [11]. The underlying concurrency model of actor languages forms the basis of the programming languages Erlang [7] and Scala [14] that have gained in popularity, in part due to their support for scalable concurrency. There are also implementations of actor libraries for Java.

* This work was funded partially by the projects FP7-ICT-610582, TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465.

Termination analysis of concurrent and distributed systems is receiving considerable attention [17,2,9]. The main challenge is in handling *shared-memory* concurrent programs. This is because, when execution interleaves from one task to another, the shared-memory may be modified by the interleaved task. The modifications will affect the behavior of the program and, in particular, can change its termination behavior and its resource consumption. Inspired by the rely-guarantee style of reasoning used for compositional verification [12] and analysis [9] of thread-based concurrent programs, we present a novel termination analysis for concurrent objects which assumes a *property* on the global state in order to prove termination of a loop and, then, proves that this property holds. The property we propose to prove is the *finiteness* of the shared-data involved in the termination proof, i.e., proving that such shared-memory is updated a finite number of times. Our method is based on a circular style of reasoning since the finiteness assumptions are proved by proving termination of the loops in which that shared-memory is modified. Crucial for accuracy is the use of the information inferred by a *may-happen-in-parallel* (MHP) analysis [4], which allows us to restrict the set of program points on which the property has to be proved to those that may actually interleave its execution with the considered loop.

Besides termination, we also are able to apply this style of reasoning in order to infer the resource consumption (or cost) of executing the concurrent program. The results of our termination analysis already provide useful information for cost: if the program is terminating, we know that the size of all data is bounded. Thus, we can give cost bounds in terms of the maximum and/or minimum values that the involved data can reach. Still, we need novel techniques to infer upper bounds on the number of iterations of loops whose execution might interleave with instructions that update the shared memory. We provide a novel approach which is based on the combination of *local* ranking functions (i.e., ranking functions obtained by ignoring the concurrent interleaving behaviors) with upper bounds on the *number of visits* to the instructions which update the shared memory. As in the case of the termination analysis, an auxiliary MHP analysis is used to restrict the set of points whose visits have to be counted to those that indeed may interleave. To the best of our knowledge this is the first approach to infer the cost of loops with concurrent interleavings.

Our analysis has been implemented, and its termination component is already fully integrated in COSTABS [2], a COSt and Termination analyzer for concurrent objects. Experimental evaluation of the termination analysis has been performed on a case study developed by Fredhopper® and several other smaller applications. Preliminary results are promising in both the accuracy and efficiency of the analysis.

The rest of the paper is organized as follows. Sec. 2 contains preliminaries about the language, termination and cost. Sec. 3 and 4 explains the rely-guarantee termination and cost analysis, respectively. Sec. 5 contains the preliminary evaluation of the analyses. Finally, Sec. 6 presents the conclusions and related work.

2 Concurrency Model, Termination and Cost

This section presents the syntax and concurrency model of the concurrent objects language, which is basically the same as [15,2]. A *program* consists of a set of classes, each of them can define a set of fields, and a set of methods. The notation $\bar{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. The set of types includes the classes and the set of *future* variable types *fut*(T). *Pure* expressions *pu* (i.e., functional expressions that do not access the shared memory) and primitive types are standard and omitted. The abstract syntax of class declarations CL, method declarations M, types T, variables V, and statements s is:

CL ::=**class** C $\{\bar{T}\ \bar{f}; \bar{M}\}$ M ::=T $m(\bar{T}\ \bar{x})\{s;$ **return** $p;\}$ V ::=x | **this**.f
s ::=$s; s$ | $x = e$ | $V = x$ | **await** $V?$ | **if** p **then** s **else** s | **while** p **do** s
e ::=**new** $C(\bar{V})$ | $V!m(\bar{V})$ | pu T ::=C | *fut*(T)

As in the actor-model, the main idea is that control and data are encapsulated within the notion of concurrent object. Thus each object encapsulates a *local heap* which stores the data that is *shared* within the object. Fields are always accessed using the **this** object, and any other object can only access such fields through method calls. We assume that every method ends with a **return** instruction. The concurrency model is as follows. Each object has a lock that is shared by all tasks that belong to the object. Data synchronization is by means of future variables: An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}x!m(\bar{z})$ such that **await** $y?$ is executed only when the future variable y is available (i.e., the task is finished). In the meantime, the object's lock can be released and some other *pending* task on that object can take it. W.l.o.g, we assume that all methods in a program have different names.

A *program state* St is a set $St = \texttt{Ob} \cup \texttt{T}$ where $\texttt{Ob}$ is the set of all created objects, and $\texttt{T}$ is the set of all created tasks. An *object* is a term $ob(o, a, lk)$ where o is the object identifier, a is a mapping from the object fields to their values, and lk the identifier of the *active task* that holds the object's lock or $\perp$ if the object's lock is free. Only one task can be *active* (running) in each object and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $tsk(t, m, o, l, s)$ where t is a unique task identifier, m is the method name executing in the task, o identifies the object to which the task belongs, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created objects and tasks never disappear from the state. Complete semantic rules can be found in the extended version of this paper [5].

Example 1. Figure 1 shows some simple examples which will illustrate different aspects of our analysis. We have an interface Task, and a class TaskQueue which implements a queue of tasks to which one can add a single task using method AddTask or a list of tasks using method AddTasks. The loop that adds the tasks invokes asynchronously method AddTask and then awaits for its termination at Line 11 (L11 for short). We use the predefined generic type List<E> with the

```
1 Class TaskQueue{
2 List<Task> pending=Nil;
3 void AddTask(Task tk){
4  pending= appendright(pending,tk);
5 }
6 void AddTasks(List<Task> list){
7  while (list != Nil) {
8    Task tk = head(list);
9    pending = tail(list);
10   Fut f=this!AddTask(tk);
11   await f?;}
12 }
13 void ConsumeAsync(){
14  while (pending != Nil) {
15   Task tk = head(pending);
16   pending = tail(pending);
17   Fut f=tk!start();}
18 }
19 void ConsumeSync(){
20  while (pending != Nil) {
21   Task tk = head(pending);
22   pending = tail(pending);
23   Fut f=tk!start();
24   await f?;}
25 }} //end class TaskQueue
26 Interface Task {void start();}
27 //implementations of main methods
28 main1(List<Task> l){
29   TaskQueue q=new TaskQueue();
30   q!AddTasks(l);
31   q!ConsumeAsync();
32 }
33 main2(List<Task> l){
34   TaskQueue q= new TaskQueue();
35   Fut f=q!AddTasks(l);
36   await f?;
37   q!ConsumeSync();
38 }
39 main3(List<Task> l){
40   TaskQueue q= new TaskQueue();
41   q!AddTasks(l);
42   q!ConsumeSync();
43 }
44 main4(List<Task> l){
45   TaskQueue q= new TaskQueue();
46    while (true){
47      Fut x=q!AddTasks(l);
48      Fut y=q!ConsumeSync();
49      await x?;
50      await y?;}
51 }
```

Fig. 1. Simple examples for termination and cost

usual operations appendright to add an element of type <E> to the end of the list, head to get the element in the head of the list and tail to get the remaining elements. These operations are performed on pure data (i.e., data that possibly contains references but does not access the shared memory) and are executed sequentially. The class has two other methods, ConsumeAsync and ConsumeSync, to consume the tasks inside the queue. The former method starts all tasks (L17) concurrently. Instead, method ConsumeSync executes each task synchronously. It releases the processor and waits until the task is finished at L24. In the rightmost column, there are four implementations of main methods which are defined outside the classes. Here we show some execution steps from main3:

$St_1 \equiv \{obj(0, f, 0)\ tsk(0, \mathsf{main3}, 0, l, \mathsf{q{=}new\ TaskQueue();...})\} \xrightarrow{new}$

$St_2 \equiv \{obj(0, f, 0)\ obj(1, f_1, \bot)\ tsk(0, \mathsf{main3}, 0, l', \mathsf{q!AddTasks(l);...})\} \xrightarrow{async-call}$

$St_3 \equiv \{obj(0, f, 0)\ obj(1, f_1, \bot)\ tsk(0, \mathsf{main3}, 0, l', \mathsf{q!ConsumeSync(1);...})$
$tsk(1, \mathsf{AddTasks}, 1, l'', \mathsf{while(list!= Nil);...})\} \xrightarrow{async-call}$

$St_4 \equiv \{obj(0, f, 0)\ obj(1, f_1, \bot)\ tsk(0, \mathsf{main3}, 0, l', \mathsf{return;})\ tsk(1, \mathsf{AddTasks}, 1, l'', ...)$
$tsk(2, \mathsf{ConsumeSync}, 1, l''', \mathsf{while(pending!= Nil);...})\} \xrightarrow{return} \xrightarrow{activate}$

$St_5 \equiv \{obj(0, f, \bot)\ obj(1, f_1, 2)\ tsk(0..)\ tsk(1..)\ tsk(2..)\}$

Observe that the execution of **new** at St_1 creates the object identified by 1. Then, the executions of the asynchronous calls at St_2 and St_3 spawn new tasks on ob-

ject 1 identified by 1 and 2, respectively. In St_4, we perform two steps, first the execution of task 0 terminates (executes return) and object 0 becomes idle, next object 1 (which was idle) selects task 2 for execution. Note that as scheduling is non-deterministic any of both pending tasks (1 or 2) could have been selected.

2.1 Termination and Cost

Traces take the form $t \equiv St_0 \rightarrow^{b_0} \cdots \rightarrow^{b_{n-1}} St_n$, where St_0 is an initial state in which only the main method is available and the superscript b_i is the instruction that is executed in the step. A trace is *complete* if it cannot continue from St_n(not taking into account spurious cycles of take-release an object's lock). A trace is *finished* if every task in the configuration $tsk(t, m, o, l, s) \in \mathtt{T}$ is finished $s = \epsilon(v)$). If a trace is complete but not finished, the trace must be *deadlocked*. Deadlocks happen when several tasks are awaiting for each other to terminate and remain blocked. Deadlock is different from non-termination, as non-terminating traces keep on consuming instructions. As we have seen, since we have no assumptions on scheduling, from a given state there may be several possible *non-deterministic* execution steps that can be taken. We say that a program is *terminating* if all possible traces from the initial state are complete.

When measuring the cost, different metrics can be considered. A cost model is a function $\mathcal{M} : Ins \mapsto \mathbb{R}^+$ which maps instructions built using the grammar above to positive real numbers and, in this way, it defines the considered metrics. The cost of an execution step is defined as $\mathcal{M}(St \rightarrow^b St') = \mathcal{M}(b)$, i.e., the cost of the instruction applied in the step. The cost of a trace is the sum of the costs of all its execution steps. The cost of executing a program is the *maximum* of the costs of all possible traces from the initial state. We aim at inferring an *upper bound* on the cost of executing a program P for the defined cost model, denoted $\mathtt{UB}_P$, which is larger than or equal to that maximum.

Example 2. A cost model that counts the number of instructions is defined as $\mathcal{M}_{inst}(b) = 1$ where b is any instruction of the grammar. A cost model that counts the number of visits to a method m is defined as $\mathcal{M}_{visits\_m}(b) = 1$ if $b = x!m(\bar{z})$ and 0 otherwise. Consider the partial trace of Ex. 1. By applying $\mathcal{M}_{inst}$ we get 4 executed instructions (as the application of Activate does not involve any instruction) and if we count $\mathcal{M}_{visits\_ConsumSync}$ we obtain 1.

3 Termination Analysis

This section gives first in Sec. 3.1 the intuition behind our method, then it presents the termination algorithm in Sec. 3.2, and finally it provides the results that we need for its application in cost analysis in Sec. 3.3.

3.1 Basic Reasoning

Our starting point is an analysis [2] that infers the termination (and resource consumption) of concurrent programs by losing all information on the shared-memory at "processor release points" (i.e., at the points in which the processor can switch the execution to another task because of an **await** instruction or a method return). Alternatively, instead of losing all information, it can also use

monitor invariants (provided by the user) to force some assumptions on the shared-memory. In the latter alternative, the correctness of the analysis depends on the correctness of the provided invariants (the analysis does not infer nor prove them correct). Let us show the kind of problems that [2] can and cannot solve. Consider the first three implementations of main methods:

- main1 creates a TaskQueue q, adds the list of tasks received as input parameter to it, and executes ConsumeAsync. It is not guaranteed that the tasks are added to the queue when ConsumeAsync starts to execute because, as the call at L30 is not synchronized, the processor can be released at L11 and the call at L31 can start to execute. This is not a problem for termination, since ConsumeAsync is executed without releasing the processor. Hence, the method of [2] can prove all methods terminating.
- in main2 the addition of tasks (i.e., the call to AddTasks at L35) is guaranteed to be terminated when ConsumeSync starts to execute due to the use of **await** at L36. However, the difficulty is that ConsumeSync contains a release point. The method of [2] fails to prove termination because at this release point pending is lost. The key is to detect that there are no concurrent interleavings at L24 in this loop by means of an auxiliary MHP analysis.
- main3 has a loop with concurrent interleavings since `ConsumeSync` is called without waiting for completion of `AddTasks`. Thus, some tasks can be added to the list of pending tasks in the middle of the execution of `ConsumeSync`, resulting in a different ordering in which tasks are executed, or even can be added when `ConsumeSync` has finished and hence start will not be executed at all on them. Proving termination requires developing novel techniques.

Our reasoning is at the level of the strongly connected components (SCCs), denoted $\langle S_1, \ldots, S_n \rangle$, in which the code to be analyzed is split. For each method m, we have an SCC named S_m and for each loop (in the methods) starting at Lx we have an SCC named S_x. The analysis starting from main2 must consider the SCCs: $\langle S_{\texttt{main2}}, S_{\texttt{AddTasks}}, S_7, S_{\texttt{AddTask}}, S_{\texttt{ConsumeSync}}, S_{20} \rangle$. For simplifying the presentation, we assume that each recursive SCC has a single cut-point (in the corresponding CFG). Moreover, the cut-point is assumed to be the entry of the SCC. In such case, an SCC can be viewed as a simple while loop (i.e., without nested loops) with several possible paths in its body. Nested loops can be transformed into this form, by viewing the inner loops as separate procedures that are called from the outer ones. This, however, cannot be done for complex mutual recursions which are rare in our context. The purpose of this assumption is to simplify the way we count the number of visits to a given program point in Sec. 4.

In order to use the techniques of [2] as a black-box, in what follows, we assume that `seq_termin`(S, F) is a basic termination analysis procedure that receives an SCC S and a set of fields F, and works as follows: (1) given a function *fields* that returns the set of fields accessed in the given scope, for any $f \in \mathit{fields}(S) \setminus F$, it adds the instruction $f = *$ at each release point of S; (2) it tries to proves termination of the instrumented code using an off-the-shelf termination analyzer for sequential code; and (3) it returns the result. We assume that `seq_termin` ignores calls to SCCs transitively invoked from the considered scope S, assumes nothing about their return values, and ignores the instruction **await**.

Algorithm 1 MHP-based Termination Analysis

```
1:  function TERMINATES(S,SSet)
2:    if S ∈ SSet then return false
3:    if seq_termin(S, ∅) then return true
4:    F = select_fields(S)
5:    if (not seq_termin(S, F)) then return false
6:    RP = release_points(S)
7:    MP = MHP_pairs(RP)
8:    I = field_updates(MP, F)
9:    DepSet = extract_sccs(I)
10:   for each S' ∈ DepSet do
11:     if (not TERMINATES(S', SSet ∪ {S}) then return false
12:   return true
```

Observation 1 (finiteness assumption) *If S terminates under the assumption that a set of fields F are not modified at the release points of S, then S also terminates if they are modified a finite number of times.*

The intuition behind our observation is as follows. Since the fields are modified finitely, then we will eventually reach a state from which that state on they are not modified. From that state, we cannot have non-termination since we know that S terminates if the fields are not modified. Moreover, one can construct a lexicographical ranking function [8] that witnesses the termination of S.

Example 3. Consider the following two loops:

S_1

```
52 while ( f > 0 ) {
53   x = g();
54   await x?;
55   f−−; }
56
```

S_2

```
57 while ( m > 0 ) {
58   x = g();
59   await x?;
60   f=∗;
61   m−−; }
```

and assume that S_1 and S_2 are the only running processes. Their execution might interleave since both loops have a release point. We let f be a shared variable, m a local variable, and we ignore the behavior of method g. It is easy to see that (a) S_1 terminates under the assumption that f does not change at the release point (L54), and that $RF_1(m, f) = f$ is a ranking function that witnesses its termination; and (b) S_2 terminates without any assumption and $RF_2(m, f) = m$ is a ranking function that witnesses its termination. Since S_2 terminates, we know that f is modified a finite number of times at the release point of S_1 and thus, according to Observation 1, S_1 terminates when running in parallel with S_2. The lexicographical ranking function $RF_3(f, m) = \langle m, f \rangle$ is a witness of the termination of S_1.

3.2 Termination Algorithm

Algorithm 1 presents the main components of our termination algorithm, defined by means of function TERMINATES. The first parameter S is an SCC that we want to prove terminating, and the second one *SSet* includes the SCCs whose

termination requires the termination of S. The role of the second parameter is to detect circular dependencies. In order to prove that a program P terminates, we prove that all its SCCs terminate by calling TERMINATES$(S, \emptyset)$ on each one of them. Let us explain the different lines of the algorithm:

1. At Line 2, if S is in the set $SSet$, then a circular dependency has been detected, i.e., the termination of S depends on the termination of S itself. In such case the algorithm returns **false** (since we cannot handle such cases).
2. At Line 3, it first tries to prove termination of S without any assumption on the fields, i.e., assuming that their values are lost at release points. If it succeeds, then it returns **true**. Otherwise, in the next lines it will try to prove termination w.r.t. some finiteness assumptions on the fields.
3. At Line 4, it selects a set of fields F and, at Line 5, it tries to prove that S terminates when assuming that fields from F keep their values at the release points. If it fails, then it returns **false**. Otherwise, in the next lines it will try to prove that these fields are modified finitely in order to apply Observation 1. The simplest strategy for constructing F (which is the one implemented in our system) is to include all fields used in S. This can also be refined to select only those that might affect the termination of S (using some dependency analysis or heuristics).
4. At this point the algorithm identifies all instructions that might modify a field from F while S is waiting at a release point. This is done as follows: at Line 6 it constructs the set RP of all release points in S; at Line 7 it constructs the set MP of all program points that may run in parallel with program points in RP (this is provided by an auxiliary MHP analysis [4]); and at Line 8 it remains with $I \subseteq MP$ that actually update a field in F.
5. At Line 9, it constructs a set $DepSet$ of *all* SCCs that can reach a program point in I, i.e., those SCCs that include a program point from I or can reach one by (transitively) calling a method that includes one. Proving termination of these SCCs guarantees that each instruction in I is executed finitely, and thus the fields in F are updated finitely and the finiteness assumption holds.
6. The loop at Line 10 tries to prove that each SCC in $DepSet$ terminates. If it finds one that might not terminate, it returns **false**. In the recursive call S is added to the second parameter in order to detect circular dependencies.
7. If the algorithm reaches Line 12, then S is terminating and returns **true**.

Essentially our approach translates the concurrent program into a sequential setting using the assumptions. To define our proposal, we have focused exclusively on the finiteness assumption because of its wide applicability for proving termination of different forms of loops. Being more general requires a more complex reasoning than when handling other kinds of simpler assumptions. For instance, simpler assumptions (like checking that a field always increases or decreases its value when it is updated) can be easily handled by adding a corresponding test, after Line 8, that checks the assumption holds on the instructions in I.

Example 4. We can now prove termination of both main2 and main3. For main2, the challenge is to prove termination of `ConsumeSync` and namely of the loop that forms S_{20}. This loop depends on the field pending whose size is decreased

at each iteration. However, there is a release point in the loop's body (L24). Thus, we need to guarantee the finiteness assumption on pending at that point. The MHP analysis infers that the only other instruction that updates pending at L4 cannot happen in parallel with the release point. This can be inferred thanks to the use of **await** at L11 and L36. Therefore, the set I at Line 8 of Alg. 1 is empty and TERMINATES returns **true**. In the analysis of main3, when proving termination of $S_{\texttt{ConsumeSync}}$ we have that L4 can happen in parallel with L24 so we have to prove the *finiteness assumption* recursively. In particular, $DepSet = \{S_{\texttt{AddTask}}, S_7, S_{\texttt{AddTasks}}, S_{\texttt{main3}}\}$. Proving termination of S_7 is done directly by `seq_termin` as termination of the loop depends only on the non-shared data list. Also, $S_{\texttt{AddTask}}$, $S_{\texttt{AddTasks}}$ and $S_{\texttt{main3}}$ are proved terminating by `seq_termin` as they do not contain loops. Thus, pending can only increase up to a certain limit and the termination of $S_{\texttt{ConsumeSync}}$ and all other scopes can be guaranteed.

We can achieve further precision by replacing `extract_sccs` by a procedure `extract_mhp_sccs` which returns all SCCs that can reach a program point in I *and* that can happen in parallel with a release point in RP. A sufficient condition for an SCC to happen in parallel with a point in RP is that its entry point (entry point of while rule) might happen in parallel with a point in RP. The correctness of this enhancement is proved in [5]. The point is that with `extract_sccs` we could find loops that contain I but cannot iterate at RP. These do not have to be taken into account because during the execution of S they will be stopped in a single iteration and therefore cannot cause unboundedness in S. This happens in the next example.

Example 5. Using `extract_mhp_sccs` we can prove that ConsumeSync always terminates in the context of main4. This is true because only one instance of AddTasks is running in parallel with ConsumeSync (due to the **await**s at L49 and L50), and AddTasks is terminating. Using `extract_sccs`, we would detect that L4 is reached from S_{46} and thus, it cannot be proved bounded (due to the **while (true)**). However, the MHP analysis tells us that the **await** in L24 of ConsumeSync can run in parallel with AddTasks but not with S_{46}. This reduces the number of SCC we have to consider (removing S_{46}) and thus we can prove ConsumeSync terminating.

Proving termination of the SCCs given by `extract_mhp_sccs` guarantees that each instruction in I is executed finitely during the release points RP, and thus the fields in F are updated finitely and the finiteness assumption holds. We assume that `extract_mhp_sccs` is used in what follows. The following theorem ensures the soundness of our approach (the proof is in [5]).

Theorem 1 (soundness). *Given a program P and its set of recursive SCCs $SSet$. If, $\forall S \in SSet$,* TERMINATES$(S, \emptyset)$ *returns* ***true****, then P is terminating.*

3.3 Inferring Field-Boundedness

The termination procedure in Sec. 3 gives us an automatic technique to infer field-boundedness, i.e., knowing that field f has upper and lower bounds on the values that it can take. The *upper* (resp. *lower*) bound of a field f is denoted as f^+ (resp. f^-), and we use f^b to refer to the bounds $[f^-, f^+]$ for f.

Corollary 1. *Consider a field f. If all recursive SCCs that reach a point in which f is updated are terminating, then f is bounded.*

4 Cost Analysis

As for termination, the resource consumption (or cost) of executing a fragment of code can be affected by concurrent interleavings in the loops. Previous work [2] is not able to estimate the cost in these cases. This section proposes new techniques to bound the number of iterations of such loops and thus the cost. This requires to have first proved field-boundedness (Sec. 3.3).

4.1 Cost Analysis of Sequential Programs

Let us first provide an intuitive view of the process of inferring the cost of a program divided in SCCs $S_1, \ldots, S_n$. As an example consider this code:

```
62 main (int n, int m)
63     { int i=0; while (i<n) { i++; s2; int j=i; while (j<m) {s1; j++; }}}
```

where s_1 and s_2 represent a sequence of instructions that do not call any other SCC and do not modify the counters. This leads to one SCC for the inner loop S_1 and one SCC for the outer loop S_2. We first consider the SCC which does not call any other scope, S_1. Given a fragment of sequential code s, we can apply the cost model $\mathcal{M}$ to all instructions in s (see Sec. 2.1) and sum the result, denoted as $\mathcal{M}(s)$. Now, an upper bound on the cost of executing the SCC S_1 is $\texttt{UB}_{S_1} = \#iter{*}\mathcal{M}(body(S_1))$ where $\#iter$ is an upper bound on the number of loop iterations. For sequential programs [3], a ranking function for the loop soundly approximates $\#iter$ and can be automatically inferred. In this case, $\texttt{UB}_{S_1} = \texttt{nat}(m{-}j{+}1){*}\mathcal{M}(body(S_1))$, where function `nat` is defined as $\texttt{nat}(n) = n$ if $n \geq 0$ and 0 otherwise (it is used to avoid having negative costs [3]).

We consider now the general case in which we need to *compose* the cost of different SCCs. The point is that in order to plug the cost that we have already computed for S_1 in its calling SCC S_2, we need to *maximize* it (i.e., compute its worst case cost). Intuitively, the worst case cost is when j is 0 and hence $\texttt{UB}_{S_1}$ becomes $\texttt{nat}(m{+}1){*}\mathcal{M}(body(S_1))$. Intuitively, maximization works by first inferring an *invariant* that holds between the arguments at the initial call (main method) and at each iteration during the execution. For instance, we infer the invariant $0 \leq j \leq m_0$ which holds in S_1 where m_0 is the initial value for m. Maximizing $\texttt{UB}_{S_1}$ using the invariant results in $\texttt{nat}(m{+}1){*}\mathcal{M}(body(S_1))$. In what follows, we refer as $\texttt{max\_init}(e)$ to the maximization of an expression e using such procedure (see [3]) which we simply adopt in this paper. Thus, the upper bound for S_2 is $\texttt{UB}_{S_2} = \#iter{*}(\mathcal{M}(body(S_2)) + \texttt{max\_init}(\texttt{UB}_{S_1})) \equiv \texttt{nat}(n){*}(\mathcal{M}(body(S_2)){+}\texttt{nat}(m{+}1){*}\mathcal{M}(body(S_1)))$.

Note that if the considered SCC is not recursive, then we simply apply $\mathcal{M}$ to the sequential instructions and compose the SCCs as above. SCCs with multiple recursive calls (that lead to an exponential complexity) and loops with logarithmic complexity are treated analogously, see [3].

4.2 Basic Reasoning

In order to explain the intuition of our approach, let us first consider the sequential loop in S_1 whose termination behavior has been widely studied by the termination community (we use $*$ to ignore irrelevant code):

S_1
```
64 while (f>0){
65   f−−;
66   if (∗ & m>0)
67         { m−−;
68           f=∗;
69 }}
```
S_2
```
70 while (f>0){
71   f−−;
72   await ∗?
73 }
```
S_3
```
74 while (m>0){
75   m−−;
76   f=∗;
77 }
```

Our method is inspired by the observation that, provided the **if** statement is executed a finite number of times, an upper bound on the number of iterations of S_1 can be computed as: the maximum number of iterations of the loop ignoring the **if** statement, but assuming that such **if** statement updates the field f with its maximum value, *multiplied* by the maximum number of times that the **if** statement can be executed. Intuitively, we assume that every time the **if** statement is executed the field can be put to its maximum value and thus the loop can be executed the maximum number of times in the next iteration. Hence, $\texttt{max\_init}(f)*m$ is an upper bound for the loop, and $\texttt{max\_init}(f) = f^+$ results in the maximum value for field f (see Sec. 3.3).

We propose to apply a similar reasoning to bound the number of iterations of loops with concurrent interleavings. Assume that S_2 and S_3 are the only running processes and that the execution of the instruction at L76 that updates the field may interleave with the **await** in S_2. We have a similar behavior to the leftmost loop, though they are obviously not equivalent. Instead of having an interleaving **if**, we have an interleaving process that updates the field. Our proposal is to first bound the number of times that instruction 76 can be executed. A sound and precise bound is m. Our main observation is that, the upper bound for S_2 is the maximum number of iterations ignoring the **await**, but assuming that at this point f can take its maximum value f^+, multiplied by the maximum number of *visits* to 76. Thus, f^+*m is a sound upper bound. If we have a loop like while (f<0) {f++; await ∗?}, whose ranking function is $-f$, then the worst case cost occurs when f is set to its minimum value f^-, i.e., $\texttt{max\_init}(-f) = f^-$. Therefore, maximizing a ranking function that involves a field f is done by relying on its field bound f^b, and it may result, depending on the case, in f^+ or f^-.

Observation 2 (loop bounds) *An upper bound on the number of iterations of a loop l with interleaving instructions that update fields F is* NITER$*$(NVISITS$+1$)*:*

1. *where* NVISITS *is the number of visits to the points in which fields in F are updated and that might interleave their execution with the loop release points;*
2. *and* NITER *is the number of iterations of the loop ignoring the interleavings —maximized w.r.t. the bounds for the fields in F;*

Our analysis relies on the assumption that the number of visits (item 1) is bounded, which has been proved in Corollary 1. Given a bound on the number of loop iterations, the cost is obtained as in the sequential case, i.e., by applying the cost model to the instructions in the loop body and multiplying it by our loop bound. Thus, we only focus now on bounding the number of loop iterations.

4.3 Bounding the Number of Iterations for Loops with Interleavings

Alg. 2 presents two mutually recursive functions which allow us to infer the two items of the observation above. For each SCC S, we assume that after executing

Algorithm 2 Bounding the Number of Iterations for Loops with Interleavings

```
1: function NITER(S, SSet)
2:   if S ∈ SSet then return false
3:   if S is not recursive then return 1
4:   i = 1;
5:   for each p ∈ S_I do
6:     i = i+NVISITS(p, S_RP, SSet ∪ S)
7:   return max_init(S_RF)*i
8: function NVISITS(p, RP, SSet)
9:   V_p= 0;
10:  P= mhp_reachable_paths(p,RP);
11:  for each ⟨S_1, ..., S_n⟩ in P do
12:    V_aux=1;
13:    for i = 1 to n do
14:      V_aux = V_aux*NITER(S_i, SSet)
15:    V_p = V_p+V_aux
16:  return V_p
```

Alg. 1 we have the following information: the set RP computed at Line 6, denoted as S_{RP}; the set I computed at Line 8, denoted as S_I; and a (linear) ranking function computed by the `seq_termin` at Lines 3 and 5, denoted as S_{RF}. If S was proved terminating at Line 3 (i.e., losing the fields), we assume that S_I and S_{RP} are empty. Function NITER receives an SCC S whose number of iterations is to be bounded and a set of SCCs $SSet$ which, as before, is initially empty and allows us to detect cyclic dependencies (Line 2). As the number of SCCs is finite, termination is guaranteed. If the SCC S is not recursive, it simply returns one (Line 3). Otherwise, the number of iterations in the SCC can be bound by the maximization of the local ranking function, multiplied by the maximum number of visits to all the points that update the fields (Line 7) *and* that may happen in parallel with S_{RP} (to this end we pass S_{RP} as parameter to NVISITS). As mentioned in Sec. 4.1, function `max_init` maximizes the received expression w.r.t. the input parameters of the entry method (often main), and the field bounds f^b are used for maximizing the fields.

Function NVISITS receives a program point p, a set of release points RP, and infers an upper bound on the number of visits to p while the program is waiting at a point of RP. We first compute the multiset of reachable paths to p. Each path is of the form $\langle S_1, \ldots, S_n \rangle$, i.e., it is a sequence of SCCs which reach the program point p. For each of the paths (Line 11), we traverse all the SCCs in the path (Line 13) and multiply the number of iterations of the corresponding SCC by those of the SCCs already traversed *if* the SCC might happen in parallel with the release points RP. We assume that `mhp_reachable_paths` gives us only those SCC that may happen in parallel with the release points RP passed as parameters. The number of visits from each of the paths is accumulated to the paths that have been already accounted (Line 15).

Example 6. Let us consider method ConsumeSync invoked from main3. We want to compute NITER($S_{20}, \emptyset$). Alg. 1 gives us that the local ranking function is $RF = \mathit{length}(\mathsf{pending})$ and that the program point 4 may happen in parallel with the release point 24 and update the field pending. Hence, we need to compute NVISITS($4, \{24\}, \{S_{20}\}$). We first compute the reachable paths to 4, which gives us the only element $\langle S_{\texttt{AddTask}}, S_7, S_{\texttt{AddTasks}} \rangle$. Note that $S_{\texttt{main3}}$ is not included in the path because its entry point cannot happen in parallel with 24. We start by com-

puting NITER($S_{\texttt{AddTask}}$, $\{S_{20}\}$), since $S_{\texttt{AddTask}}$ is not recursive, we simply return 1 which is multiplied at Line 14 of Alg. 2 by the initial value for V_{aux} (which is 1). The next iteration of the **for** loop at Line 13 invokes NITER(S_7, $\{S_{20}, S_{\texttt{AddTask}}\}$). In this case, by Alg. 1, we have the local ranking function *length*(list) and that the set of points at which list is updated is empty. The maximization of *length*(list) returns it in terms of the initial parameters of main3, i.e., *length*(l). This value is multiplied at Line 14 by 1 (previous value of V_{aux}). Finally, we compute NITER($S_{\texttt{AddTasks}}$, $\{S_7, S_{20}, S_{\texttt{AddTask}}\}$) that, as it is not recursive, simply returns 1. The execution of the **for** loop at Line 13 finishes and also the execution of the **for each** loop at Line 11 and we have that NVISITS(4, $\{S_{20}\}$)= *length*(l). Thus, we can now finish the computation of NITER($S_{20}, \emptyset$) returning *length*(pending$^+$) $*$*length*(l). The upper bound for ConsumeSync when invoked from main4 can be obtained in a similar way.

The following theorem ensures the soundness of our approach. The proof can be found in [5].

Theorem 2 (soundness). *Given a recursive SCC S, the execution of* NITER($S, \emptyset$) *terminates and returns an upper bound on the number of iterations in S.*

5 Implementation and Preliminary Evaluation

We have implemented the described cost and termination analyses, although currently only the termination component is integrated within COSTABS. Our analysis can be tried online at `http://costa.ls.fi.upm.es/costabs` by enabling the option "*rely-guarantee termination analysis*". The cost analysis component will be available for its online use from the same site soon. Given a program and a selection of an entry method from which the analysis will start, the output of the analysis is a description of the SCCs (reachable from the entry) which are terminating. This section aims at performing a preliminary experimental evaluation of the accuracy and performance of our implementation, by comparing our results with those obtained by the previous version of the analyzer which loses all information on the shared-memory. For this purpose, we have analyzed a set of small and medium-sized programs, as well as one industrial case study, the *Replication System*, developed by Fredhopper®. The analyzed code for all examples can be found and tried in the above site.

Regarding the small and medium-sized examples, their number of lines of code ranges from 20 to 100 and the number of SCCs from 5 to 20. Both versions of the analyzer need less than 1 sec. to analyze each program. All terminating loops with concurrent interleavings are reported by our rely-guarantee method, improving the results of the previous analyzer. Our largest experiment is performed on the *Replication System*, a case study that provides search and merchandising IT services to e-Commerce companies, developed within the HATS project (`http://www.hats-project.eu/`). It has 2100 lines of code and 426 SCCs that need to be analyzed. The previous analyzer needs 2813 sec. and proves 420 SCCs terminating, whereas the rely-guarantee method proves 423 SCCs terminating in only 41 sec. Times are obtained as the arithmetic mean of

five runs on a Ubuntu 12.04 32-bit with Intel Core2 Quad CPU Q9550 2.83GHz and 3.4GiB of memory. The efficiency of our rely-guarantee method can be explained because it works modularly at the level the SCCs, instead of analyzing the program globally as the previous analyzer. An inspection of the three additional SCCs that have been proved terminating confirms that they indeed correspond to loops with concurrent interleavings. The reason why a simple analysis that loses the shared-memory could achieve already good results is that the (experienced) developers of the case study were aware of the risks of having loops with concurrent interleavings and they were very much avoided.

6 Conclusions and Related Work

Concurrency adds further difficulty when attempting to prove program termination and inferring resource consumption. The problem is that the analysis must consider all possible interactions between concurrently executing objects. This is challenging because processes interact in subtle ways through fields and future variables. We have proposed novel techniques to prove termination and inferring upper bounds on the number of iterations of loops with such concurrent interleavings. Our analysis benefits from an existing MHP analysis to achieve further precision [4].

Existing methods for proving termination of thread-based programs also apply a rely-guarantee or assume-guarantee style of reasoning [9,17,10]. These methods consider every thread in isolation under assumptions on its environment, thus avoiding to reason about thread interactions directly. Applying this technique to our concurrent setting could be done by assuming a property of the second object while proving the property of the first object, and then assuming the recently proved property of the first object when proving the assumed property of the second object. Although we make assumptions and then prove them, our assumptions are of a different kind, i.e., namely they are assumptions on finiteness of data, no matter on which thread (or object) they are executed. This point makes our work fundamentally different from [9]. We can still apply our method in the presence of dynamically created objects and the number of concurrency units does not need to be known a priori as in [9].

As regards the bounds on loop iterations, to the best of our knowledge, there are no other works that have attempted to infer those bounds for loops with concurrent interleavings before. There are several techniques [13,6,20] for inferring complex loop bounds for (sequential) transition systems. Our basic termination component could benefit from these techniques. Moreover, in principle, a concurrent program could be translated to a transition system that simulates all possible interleavings, which then would allow using these techniques for inferring bounds on loops with concurrent interleaving. However, we expect such translation to be far more complicated that our techniques.

Finally, as in other kinds of analyses, by making the analysis *object-sensitive* (i.e., by distinguishing between different objects of the same class) we can achieve further precision. For instance, if we add to `main3` the following two instructions `TaskQueue q1=new TaskQueue(); q1!ConsumeSync();`. The MHP analysis infers

that ConsumeSync can run in parallel with itself. When trying to solve the equations a cyclic dependency is created and both TERMINATES and NITER algorithms terminate returning **false**.

References

1. G.A. Agha. *A*ctors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *P*roc. of APLAS'11, volume 7078 of *L*NCS, pages 238–254. Springer, December 2011.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J*ournal of Automated Reasoning, 46(2):161–203, 2011.
4. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *F*ORTE'12, *L*NCS 7273, pages 35–51. Springer, 2012.
5. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings (Extended Version). Technical Report SIC 06/13, Univ. Complutense de Madrid, 2013.
6. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *P*roc. of SAS'10, volume 6337 of *L*NCS. Springer, 2010.
7. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *C*oncurrent Programming in Erlang. Prentice Hall, 1996.
8. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. volume 3576 of *L*NCS, pages 491–504. Springer, 2005.
9. B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *P*roc. of PLDI'07, pages 320–330. ACM, 2007.
10. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *C*ommun. ACM, 54(5):88–98, 2011.
11. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *P*roc. of ESOP'07, volume 4421 of *L*NCS, pages 316–330. Springer, 2007.
12. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-Modular Verification for Shared-Memory Programs. In *E*SOP'02, LNCS 2305, pages 262–277. Springer, 2002.
13. Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *P*LDI, pages 292–304. ACM, 2010.
14. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *T*heor. Comput. Sci., 410(2-3):202–220, 2009.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *P*roc. of FMCO'10 (Revised Papers), volume 6957 of *L*NCS, pages 142–164. Springer, 2012.
16. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *A*CM Trans. Softw. Eng. Meth., 14:1–41, January 2005.
17. C. Popeea and A. Rybalchenko. Compositional Termination Proofs for Multi-Threaded Programs. In *P*roc. of TACAS'12, LNCS 7214. Springer, 2012.
18. J. Schäfer and A. Poetzsch. Jcobox: Generalizing Active Objects to Concurrent Components. In *P*roc. of ECOOP'10, LNCS 6183, pages 275–299. Springer, 2010.
19. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *P*roc. of ECOOP'08, LNCS 5142, pages 104–128. Springer, 2008.
20. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *S*AS, *L*NCS 6887, pages 280–297. Springer, 2011.

Appendix C

Article *May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization*, [14]

May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization

Elvira Albert, Samir Genaim, and Pablo Gordillo

Complutense University of Madrid (UCM), Spain

Abstract. A may-happen-in-parallel (MHP) analysis computes pairs of program points that may execute *in parallel* across different distributed components. This information has been proven to be essential to infer both safety properties (e.g., deadlock freedom) and liveness properties (termination and resource boundedness) of asynchronous programs. Existing MHP analyses take advantage of the synchronization points to learn that one task has finished and thus will not happen in parallel with other tasks that are still active. Our starting point is an existing MHP analysis developed for *intra-procedural* synchronization, i.e., it only allows synchronizing with tasks that have been spawned inside the current task. This paper leverages such MHP analysis to handle *inter-procedural* synchronization, i.e., a task spawned by one task can be awaited within a different task. This is challenging because task synchronization goes beyond the boundaries of methods, and thus the inference of MHP relations requires novel extensions to capture inter-procedural dependencies. The analysis has been implemented and it can be tried online.

1 Introduction

In order to improve program performance and responsiveness, many modern programming languages and libraries promote an asynchronous programming model, in which *asynchronous* tasks can execute concurrently with their caller tasks, until their callers explicitly wait for their completion. Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables for synchronization [8, 7]. In this model, a method call m on some parameters $\overline{x}$, written as f=m($\overline{x}$), spawns an asynchronous task. Here, f is a *future variable* which allows synchronizing with the termination of the task executing m. The instruction **await** f? allows checking whether m has finished, and blocks the execution of the current task if m is still running. As concurrently-executing tasks interleave their accesses to shared memory, asynchronous programs are prone to concurrency-related errors [6]. Automatically proving safety and liveness properties still remains a challenging endeavor today.

MHP is an analysis of utmost importance to ensure both liveness and safety properties of concurrent programs. The analysis computes *MHP pairs*, which are pairs of program points whose execution might happen in parallel across different distributed components. In this fragment of code f=m(..) ;...; **await** f?; the

execution of the instructions of the asynchronous task m may happen in parallel with the instructions between the asynchronous call and the **await**. However, due to the **await** instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the **await**. This piece of information is fundamental to prove more complex properties: in [9], MHP pairs are used to discard unfeasible deadlock cycles; in [4], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. As a simple example, consider a procedure g that contains as unique instruction y=−1, where y is a global variable. The following loop y=1; **while**(i>0){i=i−y;} might not terminate if g runs in parallel with it, since g can modify y to a negative value and the loop counter will keep on increasing. However, if we can guarantee that g will not run in parallel with this code, we can ensure termination and resource-boundedness for the loop.

This paper leverages an existing MHP analysis [3] developed for intra-procedural synchronization to the more general setting of inter-procedural synchronization. This is a fundamental extension because it allows synchronizing with the termination of a task outside the scope in which the task is spawned, as it is available in most concurrent languages. In the above example, if task g is awaited outside the boundary of the method that has spawned it, the analysis of [3] assumes that it may run in parallel with the loop and hence it fails to prove termination and resource boundedness. The enhancement to inter-procedural synchronization requires the following relevant extensions to the analysis:

1. *Must-have-finished analysis* (MHF): the development of a novel MHF analysis which infers *inter-procedural dependencies* among the tasks. Such dependencies allow us to determine that, when a task finishes, those that are awaited for on it must have finished as well. The analysis is based on using Boolean logic to represent abstract states and simulate corresponding operations. The key contribution is the use of logical implication to delay the incorporation of procedure summaries until synchronization points are reached. This is challenging in the analysis of asynchronous programs.
2. *Local MHP phase*: the integration of the above MHF information in the local phase of the original MHP analysis in which methods are analyzed locally, i.e., without taking transitive calls into account. This will require the use of richer analysis information in order to consider the inter-procedural dependencies inferred in point 1 above.
3. *Global MHP phase*: the refinement of the global phase of the MHP analysis –where the information of the local MHP analysis in point 2 is composed– in order to eliminate spurious MHP pairs which appear when inter-procedural dependencies are not tracked. This will require to refine the way in which MHP pairs are computed.

We have implemented our approach in SACO [2], a static analyzer for concurrent objects which is able to infer the aforementioned liveness and safety properties. The system can be used online at `http://costa.ls.fi.upm.es/saco/web`, where the examples used in the paper are also available.

2 Language

Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables [8, 7]. It also includes conditional and loop constructs, however, conditions in these constructs are simply non-deterministic choices. Developing the analysis at such abstract level is convenient [11], since the actual computations are simply ignored in the analysis and what is actually tracked is the control flow that originates from asynchronously calling methods and synchronizing with their termination. Our implementation, however, is done for the full concurrent object-oriented language ABS [10] (see Sec. 6).

A program P is a set of methods that adhere to the following grammar:

$$M ::= m(\bar{x})\ \{s\} \qquad s ::= \epsilon \mid b; s$$
$$b \ ::= \textsf{if}\ (*)\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{while}\ (*)\ \textsf{do}\ s \mid y = m(\bar{x}) \mid \textsf{await}\ x? \mid \textsf{skip}$$

Here all variables are future variables, which are used to synchronize with the termination of the called methods. Those future variables that are used in a method but are not in its parameters are the *local future variables* of the method (thus we do not need any special instruction for declaring them). In loops and conditions, the symbol $*$ stands for non-deterministic choice (*true* or *false*). The instruction $y = m(\bar{x})$ creates a new task which executes method m, and binds the future variable y with this new task so we can synchronize with its termination later. Inter-procedural synchronization is realized in the language by passing future variables as parameters, since the method that receives the future variable can await for the termination of the associated task (created outside its scope). For simplifying the presentation, we assume that *method parameters are not modified inside each method*. For a method m, we let P_m be the set of its parameters, L_m the set of its local variables, and $V_m = P_m \cup L_m$.

The instruction **await** $x?$ blocks the execution of the current task until the task associated with x terminates. Instruction **skip** has no effect, it is simply used when abstracting from a richer language, e.g., ABS in our case, to abstract instructions such as assignments. Programs should include a method main from which the execution (and the analysis) starts. We assume that instructions are labeled with unique identifiers that we call program points. For **if** and **while** the identifier refers to the corresponding condition. We also assume that each method has an exit program point ℓ_m. We let $\texttt{ppoints}(m)$ and $\texttt{ppoints}(P)$ be the sets of program points of method m and program P, resp., I_ℓ be the instruction at program point ℓ, and $\texttt{pre}(\ell)$ be the set of program points preceding ℓ.

Next we define a formal (interleaving) operational semantics for our language. A task is of the form $tsk(tid, l, s)$ where tid is a unique identifier, l is a mapping from local variables and parameters to task identifiers, and s is a sequence of instructions. Local futures are initialized to $\perp$. A state S is a set of tasks that are executing in parallel. From a state S we can reach a state S' in one execution step, denoted $S \rightsquigarrow S'$, if S can be rewritten using one of the derivation rules of Fig. 1 as follows: if the conclusion of the rule is $A \rightsquigarrow B$ such that $A \subseteq S$ and the premise holds, then $S' = (S \setminus A) \cup B$. The meaning of the derivation rules is quite straightforward: (SKIP) advances the execution of the corresponding task to the next instruction; (IF) nondeterministically chooses between one of

$$(\textsc{skip}) \quad \frac{}{tsk(tid, l, \textbf{skip}; s) \rightsquigarrow tsk(tid, l, s)}$$

$$(\textsc{if}) \quad \frac{b \equiv \textbf{if } (*) \textbf{ then } s_1 \textbf{ else } s_2, \text{ set } s' \text{ non-deterministically to } s_1; s \text{ or } s_2; s}{tsk(tid, l, b; s) \rightsquigarrow tsk(tid, l, s')}$$

$$(\textsc{loop}) \quad \frac{b \equiv \textbf{while } (*) \textbf{ do } s_1, \text{ set } s' \text{ non-deterministically to } s_1; b; s \text{ or } s}{tsk(tid, l, b; s) \rightsquigarrow tsk(tid, l, s')}$$

$$(\textsc{call}) \quad \frac{\bar{z} \text{ are the formal parameters of } m,\ tid' \text{ is a fresh id, } l' = \{z_i \mapsto l(x_i)\}}{tsk(tid, l, y = m(\bar{x}); s) \rightsquigarrow tsk(tid, l[y \mapsto tid'], s), tsk(tid', l', body(m))}$$

$$(\textsc{await}) \quad \frac{l(x) = tid'}{tsk(tid, l, \textbf{await } x?; s), tsk(tid', l', \epsilon) \rightsquigarrow tsk(tid, l, s), tsk(tid', l', \epsilon)}$$

Fig. 1. Derivation Rules

the branches; (LOOP) nondeterministically chooses between executing the loop body or advancing to the instruction after the loop; (CALL) creates a new task with a fresh identifier tid', initializes the formal parameters $\bar{z}$ of m to those of the actual parameters $\bar{x}$, sets future variable y in the calling task to tid', so one can synchronize with its termination later (other local futrures are assumed to be $\bot$); and (AWAIT) advances to the next instruction if the task associated to x has terminated already. Note that when a task terminates, it does not disappear from the state but rather its sequence of instructions remains empty.

An execution is a sequence of states $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_n$, sometimes denoted as $S_0 \rightsquigarrow^* S_n$, where $S_0 = \{tsk(0, l, body(\mathsf{main}))\}$ is an initial state which includes a single task that corresponds to method main, and l is an empty mapping. At each step there might be several ways to move to the next state depending on the task selected, and thus executions are nondeterministic.

In what follows, given a task $tsk(tid, l, s)$, we let $pp(s)$ be the program point of the first instruction in s. When s is an empty sequence, $pp(s)$ refers to the exit program point of the corresponding method. Given a state S, we define its set of MHP pairs, i.e., the set of program points that execute in parallel in S, as $\mathcal{E}(S) = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, l_1, s_1), tsk(tid_2, l_2, s_2) \in S, tid_1 \neq tid_2\}$. The set of MHP pairs for a program P is then defined as the the set of MHP pairs of all reachable states, namely $\mathcal{E}_P = \cup\{\mathcal{E}(S_n) \mid S_0 \rightsquigarrow^* S_n\}$.

Example 1. Fig. 2 shows some simple examples in our language. Methods m_1, m_2 and m_3 are main methods and the remaining ones are auxiliary. Let us consider some steps of one possible derivation from m_2:

$\mathbf{S_0} \equiv tsk(0, \emptyset, body(\mathsf{m}_2)) \rightsquigarrow^* \mathbf{S_1} \equiv tsk(0, [x \mapsto 1], \{16, \ldots\}), tsk(1, \emptyset, body(\mathsf{f})) \rightsquigarrow^*$
$\mathbf{S_2} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{18, \ldots\}), tsk(1, \emptyset, body(\mathsf{f})), tsk(2, [w \mapsto 1], body(\mathsf{g})) \rightsquigarrow^*$
$\mathbf{S_3} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{19, \ldots\}), tsk(1, \emptyset, \epsilon), tsk(2, [w \mapsto 1], body(\mathsf{g})) \rightsquigarrow^*$
$\mathbf{S_4} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{20, \ldots\}), tsk(1, \emptyset, \epsilon), tsk(2, [w \mapsto 1], \epsilon) \rightsquigarrow \ldots$

In S_1 we execute until the asynchronous call to f which creates a new task identified as 1 and binds x to this new task. In S_2 we have executed the **skip** and the asynchronous invocation to g that adds in the new task the binding of the formal parameter w to the task identified as 1. In S_3 we proceed with the execution of the instructions in m_2 until reaching the **await** that blocks this task until g terminates. Also, in S_3 we have executed entirely f (denoted by ϵ). S_4

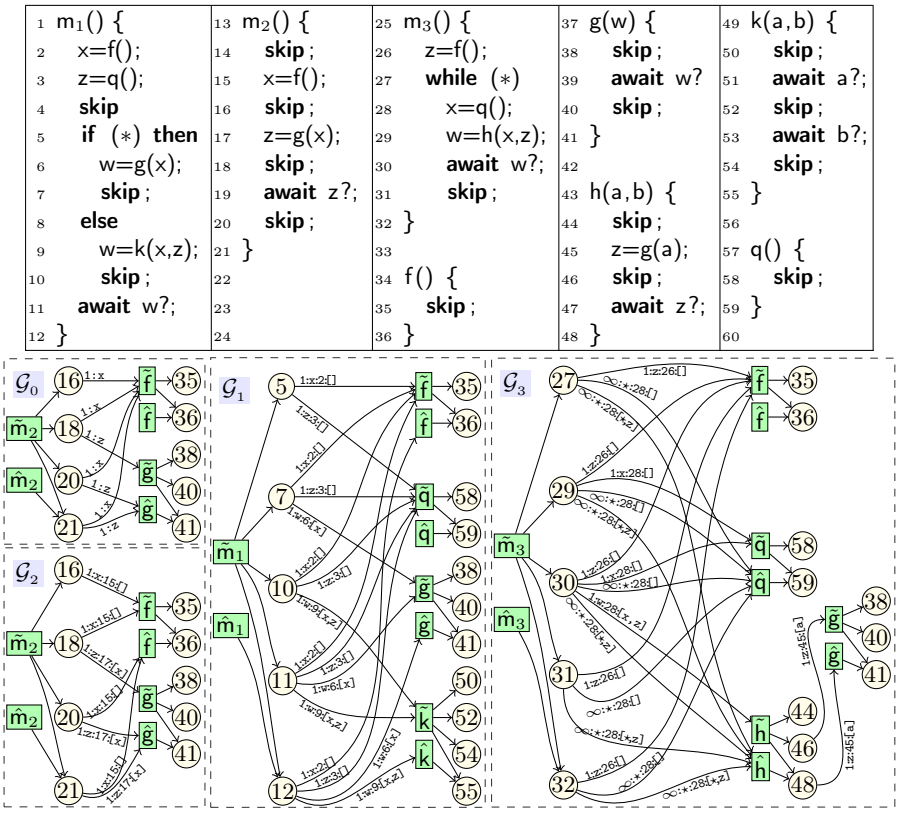


Fig. 2. (TOP) Examples for MHP analysis (m_1, m_2, m_3 are main methods). (BOTTOM) MHP graph $\mathcal{G}_i$ corresponds to analyzing m_i, and $\mathcal{G}_0$ to analyzing m_2 as in [3].

proceeds with the execution of g whose **await** can be executed since task 1 is at its exit point ϵ. We have the following MHP pairs in this fragment of the derivation, among many others: from S_1 we have (16,35) that captures that the first instruction of f executes in parallel with the instruction 16 of m2, from S_2 we have (18,35) and (18,38). The important point is that we have no pair (20,35) since when the **await** at L19 executes at S_4, it is guaranteed that f has finished. This is due to the inter-procedural dependecy at L39 of g where the task f is awaited: variable x is passed as argument to g, which allows g to synchronize with the termination of f at L39 even if f was called in a different method.

3 An Informal Account of our Method

In this section, we provide an overview of our method by explaining the analysis of m_2. Our goal is to infer precise MHP information that describes, among others, the following representative cases: (1) any program point of g cannot run in parallel with L20, because at L19 method m_2 awaits for g to terminate; (2) L35 cannot run in parallel with L20, since when waiting for the termination of g at L19 we know that f *must-have-finished* as well due to the *dependency* relation

that arises when m_2 implicitly waits for the termination of f; and (3) L35 cannot run in parallel with L40, because f *must-have-finished* due to the synchronization on the local future variable w at L39 that refers to future variable x of m_2.

Let us first informally explain which MHP information the analysis of [3] is able to infer for m_2, and identify the reasons because it fails to infer some of the desired information. The analysis of [3] is carried out in two phases: (1) each method is *analyzed separately* to infer local MHP information; and (2) the local information is used to construct a global MHP graph from which MHP pairs are extracted by checking reachability conditions among the nodes.

The local analysis infers, for each program point, a multiset of MHP atoms where each atom describes a task that might be executing in parallel when reaching that program point, but only considering tasks that have been invoked locally within the analyzed method. An atom of the form $x{:}\tilde{m}$ indicates that there might be an *active* instance of m executing at any of its program points, and is bounded to the future variable x. An atom of the form $x{:}\hat{m}$ differs from the previous one in that m must be at its exit program point, i.e., has finished executing already. For method m_2, the local MHP analysis infers, among others, $\{\mathsf{x}{:}\tilde{\mathsf{f}}\}$ for L16, $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\tilde{\mathsf{g}}\}$ for L18, and $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\hat{\mathsf{g}}\}$ for L20 and L21, because g has been awaited locally. Observe that the sets of L20 and L21 include $\mathsf{x}{:}\tilde{\mathsf{f}}$ and not $\mathsf{x}{:}\hat{\mathsf{f}}$, although we know that method f has finished already when reaching L20 and L21 (since g has finished). This information cannot be inferred by the local analysis of [3] since it is applied to each method separately, ignoring (a) transitive (non-local) calls and (b) inter-procedural synchronizations. In the sequel we let Ψ_ℓ be the result of the local MHP analysis for program point ℓ.

In the second phase, the analysis of [3] builds the MHP graph whose purpose is to capture MHP relations due to transitive calls (point (a) above). The graph $\mathcal{G}_0$ depicted in Fig. 2 for m_2 is constructed as follows: (1) every program point ℓ contributes a node labeled with ℓ – for simplicity we include only program points of interest; (2) every method m contributes two nodes $\tilde{m}$ and $\hat{m}$, where $\tilde{m}$ is connected to all program point nodes of m, to indicate that an active method can be executing at any of its program points, and $\hat{m}$ is connected only to the exit program point of m; and (3) if $x{:}\tilde{m}$ (resp. $x{:}\hat{m}$) is an atom of Ψ_ℓ with multiplicity i, we create an edge from ℓ to $\tilde{m}$ (resp. $\hat{m}$) and label it with $i{:}x$. Note that i is the multiplicity of the edge, i.e., we could copy the edge i times instead.

Roughly, the MHP pairs are obtained from $\mathcal{G}_0$ using the following principle: program points (ℓ_1, ℓ_2) might execute in parallel if there is a path from ℓ_1 to ℓ_2 or vice versa (direct MHP pair); or if there is a program point ℓ_3 such that there are paths from ℓ_3 to ℓ_1 and to ℓ_2 (indirect MHP pair), and the first edge of both paths is labeled with two different future variables. When two paths are labeled with the same future variable, it is because there is a disjunction (e.g., from an if-then-else) and only one of the paths might actually occur. Applying this principle to $\mathcal{G}_0$, we can conclude that L20 cannot execute in parallel with any program point of g, which is precise as expected, and that L20 can execute in parallel with L35 which is imprecise. This imprecision is attributed to the fact that the MHP analysis of [3] does not track inter-method synchronizations.

To overcome the imprecision, we develop a must-have-finished analysis that captures inter-method synchronizations, and use it to improve the two phases of [3]. This analysis would infer, for example, that "*when reaching* L40*, it is guaranteed that whatever task bounded to* w *has finished already*", and that "*when reaching* L20*, it is guaranteed that whatever tasks bounded to* x *and* z *have finished already*". By having this information at hand, the first phase of [3] can be improved as follows: when analyzing the effect of **await** z? at L20, we change the status of both g and f to finished, because we know that any task bounded z and x has finished already. This will require to enrich the information of the MHP atoms as follows: an MHP atom will be of the form $y{:}\ell{:}\tilde{m}(\bar{x})$ or $y{:}\ell{:}\hat{m}(\bar{x})$, where the new information ℓ and $\bar{x}$ are the calling site and the parameters passed to m. In summary, the modified first phase will infer $\{\mathsf{x}{:}15{:}\tilde{\mathsf{f}}()\}$ for L16, $\{\mathsf{x}{:}15{:}\tilde{\mathsf{f}}(), \mathsf{z}{:}17{:}\tilde{\mathsf{g}}(\mathsf{x})\}$ for L18, and $\{\mathsf{x}{:}15{:}\hat{\mathsf{f}}(), \mathsf{z}{:}17{:}\hat{\mathsf{g}}(\mathsf{x})\}$ for L20 and L21.

In the second phase of the analysis: (i) the construction of the MHP graph is modified to use the new local MHP information; and (ii) the principle used to extract MHP pairs is modified to make use to the must-have-finished information. The new MHP graph constructed for m_2 is depicted in Fig. 2 as $\mathcal{G}_2$. Observe that the labels on the edges include the new information available in the MHP atoms. Importantly, the spurious MHP information that is inferred by [3] is not included in this graph: (1) in contrast to $\mathcal{G}_0$, $\mathcal{G}_2$ does not include edges from nodes 20 and 21 to $\tilde{\mathsf{f}}$, but to $\hat{\mathsf{f}}$. This implies that L35 cannot run in parallel with L20 or L21; (2) in $\mathcal{G}_2$, we still have paths from 18 to 35 and 40, which means, if the old principle for extracting MHP pairs is used, that L35 and L40 might happen in parallel. The main point is that, using the labels on the edges, we know that the first path uses a call to f that is bounded to x, and that this same x is passed to g, using the parameter w, in the first edge of the second path. Now since the must-have-finished analysis tell us that at L40 any task bounded w is finished already, we conclude that f must be at its exit program point and thus this MHP pair is spurious (because L35 is not an exit program point).

4 Must-Have-Finished Analysis

In this section we present a novel inter-procedural Must-Have-Finished (MHF) analysis that can be used to compute, for each program point ℓ, a set of *finished future variables*, i.e., whenever ℓ is reached those variables are either not bounded to any task (i.e., have value $\perp$) or their bounded tasks are guaranteed to have terminated. We refer to such sets as MHF sets.

Example 2. The following are MHF sets for the program points of Fig. 2:

L2: {x,w,z}	L9 : {w}	L16: {z}	L26: {x,z,w}	L32: {x,w}	L41: {w}	L50: {}	L58: {}
L3: {z,w}	L10: {}	L17: {z}	L27: {x,w}	L35: {}	L44: {z}	L51: {}	L59: {}
L4: {w}	L11: {}	L18: {}	L28: {x,w}	L36: {}	L45: {z}	L52: {a}	
L5: {w}	L12: {x,w}	L19: {}	L29: {w}	L38: {}	L46: {}	L53: {a}	
L6: {w}	L14: {x,z}	L20: {x,z}	L30: {}	L39: {}	L47: {}	L54: {a,b}	
L7: {}	L15: {x,z}	L21: {x,z}	L31: {x,w}	L40: {w}	L48: {a,z}	L55: {a,b}	

At program points that correspond to method entries, all local variables (but not the parameters) are finished since they point to no task. For g: at L38 and

L39 no task is guaranteed to have finished, because the task bounded to w might be still executing; at L40 and L41, since we passed through **await** w? already, it is guaranteed that w is finished. For k: at L50 and L51 no task is guaranteed to have finished; at L52 and L53 a is finished since we already passed through **await** a?; and at L54 and L55 both a and b are finished. For m_1: at L12 both w and x are finished. Note that w is finished because of **await** w?, and x is finished due to the implicit dependency between the termination of x and w.

4.1 Definition of MHF

By carefully examining the MHF sets of Ex. 2, we can see that an analysis that simply tracks MHF sets would be imprecise. For example, since the MHF set at L11 is empty, the only information we can deduce for L12 is that w is finished. To deduce that x is finished we must track the implicit dependency between w and x. Next we define a more general MHF property that captures such dependencies, and from which we can easily compute the MHF sets.

Definition 1. *Given a program point* $\ell \in \texttt{ppoints}(P)$*, we let* $\mathcal{F}(\ell) = \{f(S_i, l) \mid S_0 \rightsquigarrow^* S_i, tsk(tid, l, s) \in S_i, pp(s) = \ell\}$ *where* $f(S, l) = \{x \mid x \in dom(l),\ l(x) = \bot \vee (l(x) = tid' \wedge tsk(tid', l', \epsilon) \in S)\}$.

Intuitively, $f(S, l)$ is the set of all future variables, from those defined in l, whose corresponding tasks are finished in S. The set $\mathcal{F}(\ell)$ considers all possible ways of reaching ℓ, and for each one it computes a corresponding set $f(S, l)$ of finished future variables. Thus, $\mathcal{F}(\ell)$ describes all possible sets of finished future variables when reaching ℓ. The set of *all* finished future variables at ℓ is then defined as $\texttt{mhf}(\ell) = \cap\{F \mid F \in \mathcal{F}(\ell)\}$, i.e., the intersection of all sets in $\mathcal{F}(\ell)$.

Example 3. The values of $\mathcal{F}(\ell)$ for selected program points from Fig. 2 are:

L5 : {{w,x,z},{w,z},{w,x},{w}}	L31: {{w,x,z},{w,x}}	L46: {{a,z},{a},{},
L11: {{w,x,z},{w,x},{x,z},{z},{x},{}}	L32: {{w,x,z},{w,x}}	{a,b,z},{a,b},{b}}
L12: {{x,w,z},{w,x}}	L35: {{}}	L48: {{a,z},{a,b,z}}
L20: {{x,z}}	L38: {{w},{}}	L52: {{a},{a,b}}
L27: {{w,x,z},{w,x}}	L40: {{w}}	L54: {{a,b}}
L30: {{w,x,z},{w,x},{x,z},{x},{z},{}}		L58: {{}}

In L5 different sets arise by considering all possible orderings in the execution of tasks f, q and m_1, but $\texttt{mhf}(L5) = \{\mathsf{w}\}$. Note that for any $F \in \mathcal{F}(11)$, if $\mathsf{w} \in F$ then $\mathsf{x} \in F$, which means that if w is finished at L11, then x must finish too.

4.2 An Analysis to Infer MHF Sets

Our goal is to infer $\texttt{mhf}(\ell)$, or a subset of it, for each $\ell \in \texttt{ppoints}(P)$. Note that any set X that over-approximates $\mathcal{F}(\ell)$, i.e., $\mathcal{F}(\ell) \subseteq X$, can be used to compute a subset of $\texttt{mhf}(\ell)$, because $\cap\{F \mid F \in X\} \subseteq \cap\{F \mid F \in \mathcal{F}(\ell)\}$. In the rest of this section we develop an analysis to over-approximate $\mathcal{F}(\ell)$. We will use Boolean formulas, whose models naturally represent MHF sets, and, moreover, Boolean connectives smoothly model the abstract execution of the different instructions.

An MHF state for the program points of a method m is a propositional formula $\Phi : V_m \mapsto \{\mathit{true}, \mathit{false}\}$ of the form $\vee_i \wedge_j c_{ij}$, where an atomic proposition c_{ij} is either x or $y \to x$ such that $x \in V_m \cup \{\mathit{true}, \mathit{false}\}$ and $y \in L_m$. Intuitively, an atomic proposition x states that x is finished, and $y \to x$ states that if y is finished then x is finished as well. Note that we do not allow the parameters of m to appear in the premise of an implication (we require $y \in L_m$). When Φ does not include any atomic proposition of the form $y \to x$ we call it monotone. Recall that $\sigma \subseteq V_m$ is a *model* of Φ, iff an assignment that maps variables from σ to *true* and other variables to *false* is a satisfying assignment for Φ. The set of all models of Φ is denoted $[\![\Phi]\!]$. The set of all MHF states for m, together with the formulas *true* and *false*, is denoted $\mathcal{A}_m$.

Example 4. Assume $V_m = \{x, y, z\}$. The Boolean formula $x \vee y$ states that either x or y or both are finished, and that z can be in any status. This information is precisely captured by the models $[\![x \vee y]\!] = \{\{x\},\{y\},\{x,y\},\{x,z\},\{y,z\},\{x,y,z\}\}$. The Boolean formula $z \wedge (x \to y)$ states that z is finished, and if x is finished then y is finished. This is reflected in $[\![z \wedge (x \to y)]\!] = \{\{z\}, \{z, y\}, \{z, x, y\}\}$ since z belongs to all models, and any model that includes x includes y as well. The formula *false* means that the corresponding program point is not reachable. The following MHF states correspond to some selected program points from Fig. 2:

Φ_5 : w	Φ_{12}: w ∧ x	Φ_{27}: w ∧ x	Φ_{31}: w ∧ x	Φ_{35}: *true*	Φ_{40}: w	Φ_{48}: a ∧ z	Φ_{54}: a ∧ b
Φ_{11}: w→x	Φ_{20}: x ∧ z	Φ_{30}: w→x	Φ_{32}: w ∧ x	Φ_{38}: *true*	Φ_{46}: z→a	Φ_{52}: a	Φ_{58}: *true*

Note that the models $[\![\Phi_\ell]\!]$ coincide with $\mathcal{F}(\ell)$ from Ex. 3.

Now that is clear how Boolean formulas represent the desired MHF information, we proceed to explain how the execution of the different instructions can be modeled with Boolean formulas. Let us first define some auxiliary operations. Given a variable x and an MHF state $\Phi \in \mathcal{A}_m$, we let $\exists x.\Phi = \Phi[x \mapsto \mathit{true}] \vee \Phi[x \mapsto \mathit{false}]$, i.e., this operation eliminates variable x from (the domain of) Φ. Note that $\exists x.\Phi \in \mathcal{A}_m$ and that $[\![\Phi]\!] \models [\![\exists x.\Phi]\!]$. For a tuple of variables $\bar{x}$ we let $\exists \bar{x}.\Phi$ be $\exists x_1.\exists x_2. \ldots .\exists x_n.\Phi$, i.e., eliminate all variables $\bar{x}$ from Φ. We also let $\bar{\exists} \bar{x}.\Phi$ stand for eliminating all variables but $\bar{x}$ from Φ. Note that if $\Phi \in \mathcal{A}_m$ and monotone, and $x \in L_m$, then $x \to \Phi$ is a formula in $\mathcal{A}_m$ as well.

Given a program point ℓ, an MHF state Φ_ℓ, and an instruction to execute I_ℓ, our aim is to compute a new MHF state, denoted $\mu(I_\ell)$, that represents the effect of executing I_ℓ within Φ_ℓ. If I_ℓ is **skip**, then clearly $\mu(I_\ell) \equiv \Phi_\ell$. If I_ℓ is an **await** x? instruction, then $\mu(I_\ell)$ is $x \wedge \Phi_\ell$, which restricts the MHF state of Φ_ℓ to those cases (i.e., models) in which x is finished. If I_ℓ is a call $y = m(\bar{x})$, where m is a method with parameters named $\bar{z}$, and, at the exit program point of m we know that the MHF state Φ_{ℓ_m} holds, then $\mu(I_\ell)$ is computed as follows:

- We compute an MHF state Φ_m that describes "what happens to tasks bounded to $\bar{x}$ when m terminates". This is done by restricting Φ_{ℓ_m} to the method parameters, and then renaming the formal parameters $\bar{z}$ to the actual parameters $\bar{x}$, i.e., $\Phi_m = (\bar{\exists} \bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]$, where $[\bar{z}/\bar{x}]$ denotes the renaming.
- Now assume that ξ is a new (future) variable to which m is bounded. Then $\xi \to \Phi_m$ states that "when m terminates, Φ_m must hold". Note that it says nothing about $\bar{x}$ if m has not terminated yet.

– Next we add $\xi \to \Phi_m$ to Φ_ℓ, eliminate (old) y since the variable is rewritten, and rename ξ to (new) y. Note that we use ξ as a temporary variable just not to conflict with the old value of y.

The above reasoning is equivalent to $(\exists y.(\Phi_\ell \wedge (\xi \to (\bar{\exists}\bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]))[\xi/y]$, and is denoted by $\oplus(\Phi_\ell, y, \Phi_{\ell_m}, \bar{x}, \bar{z})$.

Example 5. Let $\Phi_{11} = \mathsf{x} \to \mathsf{w}$ be the MHF state at L11. The effect of executing I_{11}, i.e., **await** w?, within Φ_{11} should eliminate all models that do not include w. This is done using $\mathsf{w} \wedge \Phi_{11}$ which results in $\Phi_{12} = \mathsf{w} \wedge \mathsf{x}$. Now let $\Phi_{29} = \mathsf{w}$ be the MHF state at L29. The effect of executing the instruction at L29, i.e., w=h(x,z), within Φ_{29} is defined as $\oplus(\Phi_{29}, \mathsf{w}, \Phi_{48}, \langle \mathsf{x}, \mathsf{z} \rangle, \langle \mathsf{a}, \mathsf{b} \rangle)$ and computed as follows: (1) we restrict $\Phi_{48} = \mathsf{a} \wedge \mathsf{z}$ to the method parameters $\langle \mathsf{a}, \mathsf{b} \rangle$, which results in a; (2) we rename the formal parameters $\langle \mathsf{a}, \mathsf{b} \rangle$ to the actual ones $\langle \mathsf{x}, \mathsf{z} \rangle$ which results in $\Phi_{\mathsf{h}} = \mathsf{x}$; (3) we compute $\exists \mathsf{w}.(\Phi_{29} \wedge (\xi \to \Phi_{\mathsf{h}}))$, which results in $\xi \to \mathsf{x}$; and finally (4) we rename ξ to w which results in $\Phi_{30} = \mathsf{w} \to \mathsf{x}$.

Next we define our MHF analysis by means of data-flow equations, whose solutions associate to each $\ell \in \mathtt{ppoints}(P)$ an MHF state Φ_ℓ that over-approximates $\mathcal{F}(\ell)$, i.e., $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$.

Definition 2. *The set of $\mathcal{H}_P$ of equations for a program P is defined as follows:*

1. *For each $\ell \in \mathtt{ppoints}(P)$ such that ℓ is not an entry program point, we generate the equation $\Phi_\ell = \vee\{\mu(\ell') \mid \ell' \in \mathtt{pre}(\ell)\}$;*
2. *For each $\ell \in \mathtt{ppoints}(P)$ such that ℓ is an entry program point for method $m \neq$ main, let $\{\ell_1, \ldots, l_k\}$ be the program points in which m is called, we generate $\Phi_\ell = (\vee\{TF(\Phi_{\ell_i}) \mid \ell_i \in \{\ell_1, \ldots, \ell_k\}\}) \wedge (\wedge\{x \mid x \in L_m\})$;*
3. *If ℓ is the entry program point of main, we generate $\Phi_\ell = \wedge\{x \mid x \in L_{main}\}$.*

Let us explain the meaning of each equation in the above defintion: (1) when ℓ is not a method entry, we consider each program point ℓ' that immediately precedes ℓ, compute the effect $\mu(\ell')$ of executing $I_{\ell'}$ within $\Phi_{\ell'}$, and the take the disjunction of all such states. This is precisely captured by $\Phi_\ell = \vee\{\mu(\ell') \mid \ell' \in \mathtt{pre}(\ell)\}$; (2) when ℓ is the first program point of a method $m \neq \mathsf{main}$, the most precise MHF state Φ_ℓ that we can have, ignoring how m was called, is that all local variables point to finished tasks (since they are mapped to $\bot$ when entering a method), and that the parameters might point to finished or active tasks which is precisely captured by $\Phi_\ell = \wedge\{x \mid x \in L_m\}$. But in addition we require that m has actually been called – this is the role of add $\vee\{TF(\Phi_{\ell_i}) \mid \ell_i \in \{\ell_1, \ldots, \ell_k\}\}$; (3) the equation $\Phi_\ell = \wedge\{x \mid x \in L_{\mathsf{main}}\}$, where ℓ is the entry of main, indicates that we start the execution from main.

Example 6. The following are the equations for the program points of m_3:

$\Phi_{26} = \mathsf{w} \wedge \mathsf{x} \wedge \mathsf{z}$	$\Phi_{29} = \oplus(\Phi_{28}, \mathsf{x}, \Phi_{59}, \langle\rangle, \langle\rangle)$	$\Phi_{31} = \mathsf{w} \wedge \Phi_{30}$
$\Phi_{27} = \oplus(\Phi_{26}, \mathsf{z}, \Phi_{36}, \langle\rangle, \langle\rangle) \vee \Phi_{31}$	$\Phi_{30} = \oplus(\Phi_{29}, \mathsf{w}, \Phi_{48}, \langle \mathsf{x}, \mathsf{z} \rangle, \langle \mathsf{a}, \mathsf{b} \rangle)$	$\Phi_{32} = \Phi_{27}$
$\Phi_{28} = \Phi_{27}$		

Note, for example, the circular dependency between Φ_{27} and Φ_{31} which originates from the corresponding **while** loop.

The next step is to solve $\mathcal{H}_P$, i.e., compute an MHF state Φ_ℓ, for each $\ell \in \texttt{ppoints}(P)$, such that $\mathcal{H}_P$ is satisfiable. This can be done iteratively as follows. We start from an initial solution where $\Phi_\ell = \mathit{false}$ for each $\ell \in \texttt{ppoints}(P)$. Then repeat the following until a fixed-point is reached: (1) substitute the current solution in the right hand side of the equations, and obtain new values for each Φ_ℓ; and (2) merge the new and old values of each Φ_ℓ using $\vee$. E.g, solving the equation of Ex. 6, among other equations that were omitted, results in a solution that includes, among others, the MHF states of Ex. 4. In what follows we assume that $\mathcal{H}_P$ has been solved, and we use Φ_ℓ to refer to the MHF state at ℓ in such solution.

Theorem 1. *For any program point $\ell \in \texttt{ppoints}(P)$, we have $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$.*

In the rest of this article we let $\texttt{mhf}_\alpha(\ell) = \{x \mid x \in V_m,\ \Phi_\ell \models x\}$, i.e., the set of finished future variable at ℓ that is induced by Φ_ℓ. Theorem 1 implies $\texttt{mhf}_\alpha(\ell) \subseteq \texttt{mhf}(\ell)$. Computing $\texttt{mhf}_\alpha(\ell)$ using the MHF states of Ex. 4, among others that are omitted, results exactly in the MHF sets of Ex. 2.

5 MHP Analysis

In this section we present our MHP analysis, which is based on incorporating the MHF sets of Sec. 4 into the MHP analysis of [3]. In sections 5.1 and 5.2 we describe how we modify the two phases of the original analysis, and describe the gain of precision with respect to [3] in each phase.

5.1 Local MHP

The local MHP analysis (LMHP) considers each method m separately, and for each $\ell \in \texttt{ppoints}(m)$ it infers an LMHP state that describes the tasks that might be executing when reaching ℓ (considering only tasks invoked in m). An LMHP state Ψ is a *multiset* of MHP atoms, where each atom represents a task and can be: (1) $y{:}\ell'{:}\tilde{m}(\bar{x})$, which represents an *active* task that might be at any of its program points, including the exit one, and is bounded to future variable y. Moreover, this task is an instance of method m that was called at program point ℓ' (the *calling site*) with future parameters $\bar{x}$; or (2) $y{:}\ell'{:}\hat{m}(\bar{x})$, which differs from the previous one in that the task can only be at the exit program point, i.e., it is a *finished* task. In both cases, future variables y and $\bar{x}$ can be $\star$, which is a special symbol indicating that we have no information on the future variable.

Intuitively, the MHP atoms of Ψ represent (local) tasks that are executing in parallel. However, since a variable y cannot be bounded to more than one task at the same time, atoms bounded to the same variable represent mutually exclusive tasks, i.e., cannot be executing at the same time. The same holds for atoms that use *mutually exclusive calling sites* ℓ_1 and ℓ_2 (i.e., there is no path from ℓ_1 and ℓ_2 and vice versa). The use of multisets allows including the same atom several times to represent different instances of the same method. We let $(a, i) \in \Psi$ indicate that a appears i times in Ψ. Note that i can be ∞, which happens when the atom corresponds to a calling site inside a loop, this guarantees convergence

of the analysis. Note that the MHP atoms of [3] do not use the parameters $\bar{x}$ and the calling site ℓ', since they do not benefit from such extra information.

Example 7. The following are LMHP states for some program points from Fig. 2:

L5 : {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$()}
L7 : {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:6:$\tilde{\mathsf{g}}$(x)}
L10: {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:9:$\tilde{\mathsf{k}}$(x,z)}
L11: {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:6:$\tilde{\mathsf{g}}$(x),w:9:$\tilde{\mathsf{k}}$(x,z)}
L12: {x:2:$\hat{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:6:$\hat{\mathsf{g}}$(x),w:9:$\hat{\mathsf{k}}$(x,z)}
L16: {x:15:$\tilde{\mathsf{f}}$()}
L18: {x:15:$\tilde{\mathsf{f}}$(),z:17:$\tilde{\mathsf{g}}$(x)}
L20: {x:15:$\hat{\mathsf{f}}$(),z:17:$\hat{\mathsf{g}}$(x)}
L21: {x:15:$\hat{\mathsf{f}}$(),z:17:$\hat{\mathsf{g}}$(x)}
L27: {z:26:$\tilde{\mathsf{f}}$(),(⋆:28:$\hat{\mathsf{q}}$(),∞),(⋆:29:$\hat{\mathsf{h}}$(⋆,z),∞)}
L29: L27 ∪ {x:28:$\tilde{\mathsf{q}}$()}
L30: L29 ∪ {w:29:$\tilde{\mathsf{h}}$(x,z)}
L31: {z:26:$\tilde{\mathsf{f}}$(),(⋆:28:$\hat{\mathsf{q}}$(),∞),(⋆:29:$\hat{\mathsf{h}}$(⋆,z),∞)}
L32: {z:26:$\tilde{\mathsf{f}}$(),(⋆:28:$\hat{\mathsf{q}}$(),∞),(⋆:29:$\hat{\mathsf{h}}$(⋆,z),∞)}
L44: {}
L46: {z:45:$\tilde{\mathsf{g}}$(a)}
L48: {z:45:$\hat{\mathsf{g}}$(a)}

Let us explain some of the above LMHP states. The state at L5 includes x:2:$\tilde{\mathsf{f}}$() and z:3:$\tilde{\mathsf{q}}$() for the active tasks invoked at L2 and L3. The state at L11 includes an atom for each task invoked in m1. Note that those of g and h are bounded to the same future variable w, which means that only one of them might be executing at L11, depending on which branch of the **if** statement is taken. The state at L12 includes z:3:$\tilde{\mathsf{q}}$() since q might be active at L12 if we take the **then** branch of the **if** statement, and the other atoms correspond to tasks that are finished. The state at L27 includes z:26:$\tilde{\mathsf{f}}$() for the active task invoked at L26, and ⋆:28:$\hat{\mathsf{q}}$() and ⋆:29:$\hat{\mathsf{h}}$(⋆,z) with ∞ multiplicity for the tasks created inside the loop. Note that the first parameter of h is ⋆ since x is rewritten at each iteration.

The LMHP states are inferred by a *data-flow analysis* which is defined as a solution of a set of LMHP constraints obtained by applying the following transfer function τ to the instructions. Given an LMHP state Ψ_ℓ, the effect of executing instruction I_ℓ within Ψ_ℓ, denoted by $\tau(I_\ell)$, is defined as follows:

- if I_ℓ is a call $y = m(\bar{x})$, then $\tau(I_\ell) = \Psi_\ell[y/\star] \cup \{y{:}\ell'{:}\tilde{m}(\bar{x})\}$, which replaces each occurrence of y by $\star$, since it is rewritten, and then adds a new atom $y{:}\ell{:}\tilde{m}(\bar{x})$ for the newly created task. E.g., the LMHP state of L30 in Ex. 7 is obtained from the one of L29 by adding w:29:$\tilde{\mathsf{h}}$(x,z) for the call at L29;
- if I_ℓ is **await** y?, and ℓ' is the program point after ℓ, then we mark all tasks that are bounded to a finished future variable as finished, i.e., $\tau(I_\ell)$ is obtained by turning each $z{:}\ell''{:}\tilde{m}(\bar{x}) \in \Psi_\ell$ to $z{:}\ell''{:}\hat{m}(\bar{x})$ for each $z \in \mathtt{mhf}_\alpha(\ell')$. E.g., the LMHP state of L12 in Ex. 7 is obtained from the one of L11 by turning the status of g, k, and f to finished (since w and x are finished at L12);
- otherwise, $\tau(I_\ell) = \Psi_\ell$.

The main difference w.r.t. the analysis of [3] is the treatment of **await** y?: while we use an MHF set computed using the inter-procedural MHF analysis of Sec. 4, In [3] the MHF set $\{y\}$ is used, which is obtained syntactically from the instruction. Our LMHP analysis, as [3], is defined as a solution of a set of LMHP constraints. In what follows we assume that the results of the LMHP analysis are available, and we will refer to the LMHP state of program point ℓ as Ψ_ℓ.

5.2 Global MHP

The results of the LMHP analysis are used to construct an MHP graph, from which we can compute the desired set of MHP pairs. The construction is exactly as in [3] except that we carry the new information in the MHP atoms. However, the process of extracting the MHP pairs from such graphs will be modified.

In what follows, we use $y{:}\ell{:}\check{m}(\bar{x})$ to refer to an MHP atom without specifying if it corresponds to an active or finished task, i.e., the symbol $\check{m}$ can be matched to $\tilde{m}$ or $\hat{m}$. As in [3], the nodes of the MHP graph consist of two method nodes $\tilde{m}$ and $\hat{m}$ for each method m, and a program point node ℓ for each $\ell \in \texttt{ppoints}(P)$. Edges from $\tilde{m}$ to each $\ell \in \texttt{ppoints}(m)$ indicate that when m is active, it can be executing at any program point, including the exit, but only one. An edge from $\hat{m}$ to ℓ_m indicates that when m is finished it can be only at its exit program point. The out-going edges from a program point node ℓ reflect the atoms of the LMHP state Ψ_ℓ as follows: if $(y{:}\ell'{:}\check{m}(\bar{x}), i) \in \Psi_\ell$, then there is an edge from node ℓ to node $\check{m}$ and it is labeled with $i{:}y{:}\ell'{:}\bar{x}$. These edges simply indicate which tasks might be executing in parallel when reaching ℓ, exactly as Ψ_ℓ does.

Example 8. The MHP graphs $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ in Fig. 2, correspond to methods m_1, m_2, and m_3, each analyzed together with its reachable methods. For simplicity, the graphs include only some program points of interest. Note that the out-going edges of program point nodes coincide with the LMHP states of Ex. 7.

The procedure of [3] for extracting the MHP pairs from the MHP graph of a program P, denoted $\mathcal{G}_P$, is based on the following principle: (ℓ_1, ℓ_2) is an MHP pair induced by $\mathcal{G}_P$ iff (i) $\ell_1 \rightsquigarrow \ell_2 \in \mathcal{G}_P$ or $\ell_2 \rightsquigarrow \ell_1 \in \mathcal{G}_P$; or (ii) there is a program point node ℓ_3 and paths $\ell_3 \rightsquigarrow \ell_1 \in \mathcal{G}_P$ and $\ell_3 \rightsquigarrow \ell_2 \in \mathcal{G}_P$, such that the first edges of these paths are different and they do not correspond to mutually exclusive MHP atoms, i.e., they use different future variable and do not correspond to mutually exclusive calling sites (see Sec. 5.1). Edges with multiplicity $i > 1$ represent i different edges. The first (resp. second) case is called direct (resp. indirect) MHP, see Sec. 3.

Example 9. Let us explain some of the MHP pairs induced by $\mathcal{G}_1$ of Fig. 2. Since $11 \rightsquigarrow 35 \in \mathcal{G}_1$ and $11 \rightsquigarrow 58 \in \mathcal{G}_1$ we conclude that (11,58) and (11,35) are direct MHP pairs. Moreover, since these paths originate in the same node 11, and the first edges use different future variables, we conclude that (58,35) is an indirect MHP pair. Similarly, since $11 \rightsquigarrow 38 \in \mathcal{G}_1$ and $11 \rightsquigarrow 50 \in \mathcal{G}_1$ we conclude that (11,38) and (11,50) are direct MHP pairs. However, in this case (38,50) is not an indirect MHP pair because the first edges of these paths use the same future variable w. Indeed, the calls to g and k appear in different branches of an **if** statement. To see the improvement w.r.t. to [3] note that node 12 does not have an edge to $\tilde{\mathsf{f}}$, since our MHF analysis infers that x is finished at that L12. The analysis of [3] would have an edge to $\tilde{\mathsf{f}}$ instead of $\hat{\mathsf{f}}$, and thus it produces spurious pairs such as (12,35). Similar improvements occur also in $\mathcal{G}_2$ and $\mathcal{G}_3$.

Now consider nodes 35 and 40, and note that we have $11 \rightsquigarrow 35 \in \mathcal{G}_1$ and $11 \rightsquigarrow 40 \in \mathcal{G}_1$, and moreover these paths use different future variables. Thus, we

conclude that (35,40) is an indirect MHP pair. However, carefully looking at the program we can see that this is a spurious pair, because x (to which task f is bounded) is passed to method g, as parameter w, and w is guaranteed to finish when executing **await** w? at L39. A similar behavior occurs also in $\mathcal{G}_2$ and $\mathcal{G}_3$. For example, the paths $30 \rightsquigarrow 58 \in \mathcal{G}_3$ and $30 \rightsquigarrow 40 \in \mathcal{G}_3$ induce the indirect MHP pair (58,40), which is spurious since x is passed to h at L29, as parameter a, which in turn is passed to g at L45, as parameter w, and w is guaranteed to finish when executing **await** w? at L39.

The spurious pairs in the above example show that even if we used our improved LMHP analysis when constructing the MHP graph, using the procedure of [3] to extract MHP pairs might produce spurious pairs. Next we handle this imprecision, by modifying the process of extracting the MHP pairs to have an extra condition to eliminate such spurious MHP pairs. This condition is based on identifying, for a given path $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$, which of the parameters of m are guaranteed to finish before reaching ℓ, and thus, any task that is passed to m in those parameters cannot execute in parallel with ℓ.

Definition 3. *Let p be a path $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$, $\bar{z}$ be the formal parameter of m, and I a set of parameter indices of method m. We say that I is* not alive along p *if (i) p has a single edge, and for some $i \in I$ the parameter z_i is in $\texttt{mhf}_\alpha(\ell)$; or (ii) p is of the form $\breve{m} \longrightarrow \ell_1 \overset{i:k:y:\bar{x}}{\longrightarrow} \breve{m}_1 \rightsquigarrow \ell$, and for some $i \in I$ the parameter z_i is in $\texttt{mhf}_\alpha(\ell_1)$ or $I' = \{j \mid i \in I, z_i = x_j\}$ is not alive along $\breve{m}_1 \rightsquigarrow \ell$.*

Intuitively, I is not alive along p if some parameter z_i, with $i \in I$, is finished at some point in p. Thus, any task bounded to z_i cannot execute in parallel with ℓ.

Example 10. Consider $p \equiv \tilde{g} \rightsquigarrow 40 \in \mathcal{G}_1$, and let $I = \{1\}$, then I is not alive along p since it is a path that consists of a single edge and $\texttt{w} \in \texttt{mhf}_\alpha(40)$. Now consider $\tilde{h} \rightsquigarrow 40 \in \mathcal{G}_3$, and let $I = \{1\}$, then I is not alive along p since $I' = \{1\}$ is not alive along $\tilde{g} \rightsquigarrow 40$.

The notion of "*not alive along a path*" can be used to eliminate spurious MHP pairs as follows. Consider two paths

$$p_1 \equiv \ell_3 \overset{i_1:y_1:\ell_1':\bar{w}}{\longrightarrow} \tilde{m}_1 \rightsquigarrow \ell_1 \in \mathcal{G}_P \quad \text{and} \quad p_2 \equiv \ell_3 \overset{i_2:y_2:\ell_2':\bar{x}}{\longrightarrow} \breve{m}_2 \rightsquigarrow \ell_2 \in \mathcal{G}_P$$

such that $y_1 \neq \star$, and the first node after $\tilde{m}_1$ does not correspond to the exit program point of m_1, i.e., m_1 is not finished and bounded to y_1. Define

- $F = \{y_1\} \cup \{y \mid \Phi_{\ell_3} \models y \rightarrow y_1\}$, i.e., the set of future variables at ℓ_3 such that when any of them is finished, y_1 is finished as well; and
- $I = \{i \mid y \in F,\ x_i = y\}$, i.e., the indices of the parameters of m_2 to which we pass variables from F (in p_2).

We claim that if I is not alive along p_2, then the MHP pair (ℓ_1, ℓ_2) is spurious. This is because before reaching ℓ_2, some task from F is guaranteed to terminate, and hence the one bounded to y_1, which contradicts the assumption that m_1 is not finished. In such case p_1 and p_2 are called mutually exclusive paths.

Example 11. We reconsider the spurious indirect MHP pairs of Ex. 9. Consider first (35,40), which originates from

$$p_1 \equiv 11 \overset{1:\mathsf{x}:2:[]}{\longrightarrow} \tilde{f} \rightsquigarrow 35 \in \mathcal{G}_1 \text{ and } p_2 \equiv 11 \overset{1:\mathsf{w}:6:[\mathsf{x}]}{\longrightarrow} \tilde{g} \rightsquigarrow 40.$$

We have $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, and we have seen in Ex. 10 that I is not alive along $\tilde{g} \rightsquigarrow 40 \in \mathcal{G}_1$, thus p_1 and p_2 are mutually exclusive and we eliminate this pair. Similarly, consider (58,40) which originates from

$$p_1 \equiv 30 \overset{1:\mathsf{x}:28:[]}{\longrightarrow} \tilde{q} \rightsquigarrow 58 \in \mathcal{G}_3 \text{ and } p_2 \equiv 30 \overset{1:\mathsf{w}:29:[\mathsf{x},\mathsf{z}]}{\longrightarrow} \tilde{h} \rightsquigarrow 40.$$

Again $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, and we have seen in Ex. 10 that I is not alive along $\tilde{h} \rightsquigarrow 40 \in \mathcal{G}_3$, thus p_1 and p_2 are mutually exclusive and we eliminate this pair.

Let $\tilde{\mathcal{E}}_P$ be the set of all MHP pairs obtained by applying the process of [3], modified to eliminate indirect pairs that correspond to mutually exclusive paths.

Theorem 2. $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

6 Conclusions, Implementation and Related Work

The main contribution of this work has been the enhancement of an MHP analysis that could only handle a restricted form of intra-procedural synchronization to the more general inter-procedural setting, as available in today's concurrent languages. Our analysis has a wide application scope on the inference of the main properties of concurrent programs, namely the new MHP relations are essential to infer (among others) the properties of the termination, resource usage and deadlock freedrom of programs that use inter-procedural synchronization.

Our analysis has been implemented in SACO [2], a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects, which is able to infer deadlock, termination and resource boundedness of ABS programs [10] that follow the concurrent objects paradigm. Concurrent objects are based on the notion of concurrently running objects, similar to the actor-based and active-objects approaches [12, 13]. These models take advantage of the concurrency implicit in the notion of object to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. Concurrent objects communicate via *asynchronous* method calls and use **await** instructions to synchronize with the termination of the asynchronous tasks. Therefore, the abstract model used in Sec. 2 fully captures the MHP relations arising in ABS programs.

The implementation has been built on top of the original MHP analysis in SACO. The MHF analysis has been implemented and its output has been used within the local and global phases of the MHP analysis, which have been adapted to this new input as described in the technical sections. The remaining analyses in SACO did not require any modification and now they work for inter-procedural synchronization as well. Our method can be tried online at: `http://costa.ls.fi.upm.es/saco/web` by enabling the option `Inter-Procedural Synchronization` of the `MHP` analysis in the `Settings` section. One can then

apply the MHP analysis by selecting it from the menu for the types of analyses and then clicking on `Apply`. All examples used in the paper are available in the folder `Forte15` adapted to the syntax of the ABS language. In the near future, we plan to apply our analysis to industrial case studies that are being developed in ABS but that are not ready for experimentation yet.

There is an increasing interest in asynchronous programming and in concurrent objects, and in the development of program analyses that reason on safety and liveness properties [6]. Existing MHP analyses for asynchronous programs [3, 11, 1] lose all information when future variables are used as parameters, as they do not handle inter-procedural synchronization. As a consequence, existing analysis for more advanced properties [9, 4] that rely on the MHP relations do all lose the associated analysis information on such futures. In future work we plan to study the complexity of our analysis, which we conjuncture to be in the same complexity order as [4]. In addition, we plan to study the computational complexity of deciding MHP, for our abstract models, with and without inter-procedural synchronizations in a similar way to what has been done in [5] for the problem of state reachability.

References

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Proc. of PPOPP'07*, pages 183–193. ACM, 2007.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
4. 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.
5. A. Bouajjani and M. Emmi. Analysis of Recursively Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(3):10, 2013.
6. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable Refinement Checking for Concurrent Objects. In *Proc. of POPL 2015*, pages 651–662. ACM, 2015.
7. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *ESOP'07*, LNCS 4421, pages 316–330. Springer, 2007.
8. C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.
9. 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.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO'10 (Revised Papers)*, LNCS 6957, pp. 142-164. Springer, 2012.

11. J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient may happen in parallel analysis for async-finish parallelism. In *In SAS 2012*, volume 7460, pages 5–23. Springer, 2012.
12. J. Schäfer and A. Poetzsch-Heffter. JCobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, LNCS, pages 275–299. Springer, 2010.
13. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008.

Appendix D

Article *Quantified Abstract Configurations of Distributed Systems*, [9]

Author's personal copy

DOI 10.1007/s00165-014-0321-z
BCS © 2014
Formal Aspects of Computing

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] Departamento Sistemas Informáticos y Computación (DSIC), Facultad de Informática, Universidad Complutense de Madrid, C/Prof. José García Santesmases, s/n, 28040 Madrid, Spain
[2] Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, España

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] is an example of this). In this paper, we introduce the notion of *quantified abstraction* (QA) 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.

Correspondence and offprint requests to: E. Albert, E-mail: elvira@sip.ucm.es

Published online: 06 November 2014

Author's personal copy

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.

Author's personal copy

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 Sect. 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 Sect. 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 Sect. 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, Sect. 9 summarizes the main conclusions of this article and points out several directions for future research.

Author's personal copy

2. The language

We apply our analysis to the language ABS [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

$$\texttt{class}\ C_1\ (t_1\ \mathit{fn}_1, \ldots, t_n\ \mathit{fn}_n)\quad \{M_1\ \ldots\ M_k\}$$

where each $t_i\ \mathit{fn}_i$ declares a field fn_i of type t_i, and each M_i is a *method definition*

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

where t is the type of the return value; $w_1, \ldots, w_n$ are the formal parameters with types $t_1, \ldots, t_n$; $w_{n+1}, \ldots, w_{n+p}$ are local variables with types $t_{n+1}, \ldots, t_{n+p}$; 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.

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

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 $\mathit{cobox}(c, o)$, objects $\mathit{ob}(o, a, t, t_s)$ and futures $\mathit{fut}(f, v)$. 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 $\mathit{fut}(f, v)$ 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.

Author's personal copy

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

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

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

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

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

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

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

Fig. 1. ABS language semantics

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 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 $\{ob(o_\epsilon, \bot, idle, \{\langle \bot, (f_\epsilon = this!\mathsf{main}(\bar{v}))\rangle\}), cobox(c_\epsilon, idle), fut(f_\epsilon, \bot)\}$. 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 $P(\overline{x})$, multiple (possibly infinite) fully expanded traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible fully expanded traces for $P(\overline{x})$.

Author's personal copy


```
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

Example 2.1 Our running example in Fig. 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 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 Fig. 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). ■

Author's personal copy

$ob(o_\epsilon, [cobox \mapsto c_\epsilon], \langle [\mathsf{s} \mapsto \bot, ret \mapsto f_\epsilon], (\mathsf{Server\ s} = \textbf{newcog}\ \mathsf{Server(null)};\ldots)\rangle, \emptyset)\ cobox(c_\epsilon, o_\epsilon)\ fut(f_\epsilon, \bot)$

(1) $\leadsto_{o_\epsilon}^{(\text{new-cobox})}$ $ob(o_\epsilon, [cobox \mapsto c_\epsilon], \langle [\mathsf{s} \mapsto o_1, ret \mapsto f_\epsilon], (\mathsf{s!start(n,m)};\ldots)\rangle, \emptyset)\ ob(o_1, [\mathsf{dao} \mapsto null, \mathsf{cobox} \mapsto c_1], idle, \emptyset)$
$cobox(c_\epsilon, o_\epsilon)\ cobox(c_1, idle)\ fut(f_\epsilon, \bot)$

(2) $\leadsto_{o_\epsilon}^{(\text{async-call})}$ $ob(o_\epsilon, [cobox \mapsto c_\epsilon], \langle [\mathsf{s} \mapsto o_1, ret \mapsto f_\epsilon], \textbf{return;}\rangle, \emptyset)$
$ob(o_1, [\mathsf{dao} \mapsto null, cobox \mapsto c_1], idle, \{\langle [\mathsf{n} \mapsto 2, \mathsf{m} \mapsto 2, this \mapsto o_1, \mathsf{f} \mapsto \bot, ret \mapsto f_1], (\mathsf{Fut\ f<}\textbf{void}\mathsf{>=}\textbf{this}\mathsf{!initDAO();}\ldots)\rangle\})$
$cobox(c_\epsilon, o_\epsilon)\ cobox(c_1, idle)\ fut(f_\epsilon, \bot)\ fut(f_1, \bot)$

(3) $\leadsto_{o_\epsilon}^{(\text{return})}$ $ob(o_\epsilon, [cobox \mapsto c_\epsilon], idle, \emptyset)$
$ob(o_1, [\mathsf{dao} \mapsto null, cobox \mapsto c_1], idle, \{\langle [\mathsf{n} \mapsto 2, \mathsf{m} \mapsto 2, this \mapsto o_1, \mathsf{f} \mapsto \bot, ret \mapsto f_1], (\mathsf{Fut\ f<}\textbf{void}\mathsf{>=}\textbf{this}\mathsf{!initDAO();}\ldots)\rangle\})$
$cobox(c_\epsilon, idle)\ cobox(c_1, idle)\ fut(f_\epsilon, null)\ fut(f_1, \bot)$

(4) $\leadsto_{o_1}^{(\text{activate})}$ $ob(o_\epsilon, [cobox \mapsto c_\epsilon], idle, \emptyset)$
$ob(o_1, [\mathsf{dao} \mapsto null, cobox \mapsto c_1], \langle [\mathsf{n} \mapsto 2, \mathsf{m} \mapsto 2, this \mapsto o_1, \mathsf{f} \mapsto \bot, ret \mapsto f_1], (\mathsf{Fut\ f<}\textbf{void}\mathsf{>} = \textbf{this}\mathsf{!initDAO();}\ldots)\rangle, \emptyset)$
$cobox(c_\epsilon, idle)\ cobox(c_1, o_1)\ fut(f_\epsilon, null)\ fut(f_1, \bot)$

(5) $\leadsto_{o_1}^{(\text{async-call})}$ $ob(o_\epsilon, [cobox \mapsto c_\epsilon], idle, \emptyset)$
$ob(o_1, [\mathsf{dao} \mapsto null, cobox \mapsto c_1], \langle [\mathsf{n} \mapsto 2, \mathsf{m} \mapsto 2, this \mapsto o_1, \mathsf{f} \mapsto f_2, ret \mapsto f_1], (\textbf{await}\ \mathsf{f?;}\ldots)\rangle,$
$\{\langle [this \mapsto o_1, \mathsf{f} \mapsto \bot, ret \mapsto f_2], (\textbf{this}.\mathsf{dao} = \textbf{new}\ \mathsf{DAO(null)})\rangle\})$
$cobox(c_\epsilon, idle)\ cobox(c_1, o_1)\ fut(f_\epsilon, null)\ fut(f_1, \bot)\ fut(f_2, \bot)$

(6) $\leadsto_{o_1}^{(\text{await-f})}$ $ob(o_\epsilon, [cobox \mapsto c_\epsilon], idle, \emptyset)$
$ob(o_1, [\mathsf{dao} \mapsto null, cobox \mapsto c_1], idle,$
$\{\langle [this \mapsto o_1, \mathsf{f} \mapsto \bot, ret \mapsto f_2], (\textbf{this}.\mathsf{dao} = \textbf{new}\ \mathsf{DAO(null)})\rangle, \langle [\mathsf{n} \mapsto 2, \mathsf{m} \mapsto 2, this \mapsto o_1, \mathsf{f} \mapsto f_2, ret \mapsto f_1],$
$(\textbf{await}\ \mathsf{f?;}\ldots)\rangle\})$
$cobox(c_\epsilon, idle)\ cobox(c_1, idle)\ fut(f_\epsilon, null)\ fut(f_1, \bot)\ fut(f_2, \bot)$

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

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]. 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.

Author's personal copy

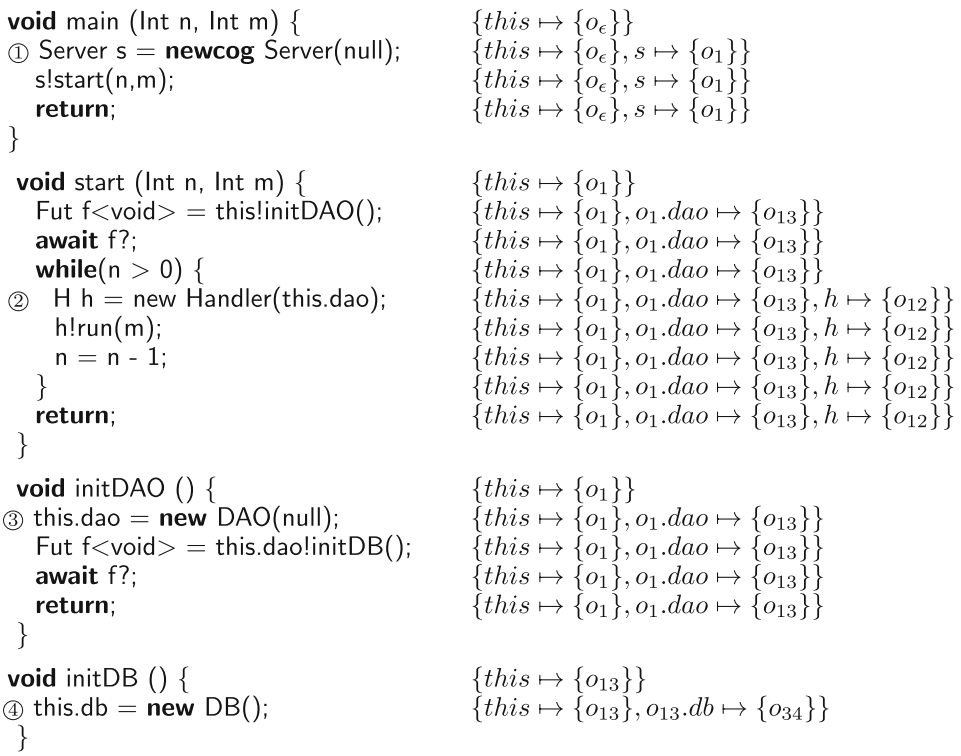


Fig. 4. Fragment of points-to analysis with $k = 2$ applied to the running example

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 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 Sect. 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 $\mid s \mid$ 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 $\mid ab \ldots cd \mid \leq k$, 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.

Author's personal copy

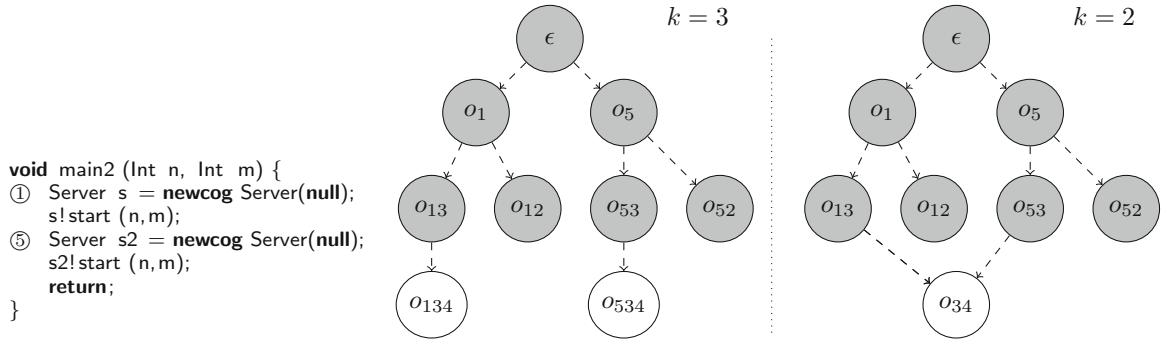


Fig. 5. Abstract configurations example with different values of k

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 $13 \oplus_2 4$ 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 Fig. 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 $o' \neq o_\lambda$.

Author's personal copy

Given a trace step $S_i \rightsquigarrow^{b}_{o_l} S_{i+1}$, we will use $\mathcal{M}(S_i \rightsquigarrow^{b}_{o_l} S_{i+1})$ 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 $cost_P(t, o_l, \mathcal{M}) = \sum_{s \in steps(t)} \mathcal{M}(s) \mid_{\{name(o_l)\}}$, where we use $\mathcal{M}(s) \mid_{\mathcal{N}}$ 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* $UB^{\mathcal{M}}_P(\overline{x})$ that is an expression of the form $c_1 \cdot e_1 + \cdots + c_n \cdot e_n$ 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 $UB^{\mathcal{M}^{\mathcal{I}}}_P(\overline{x})$ is an expression of the form $c(o_{\lambda_1}) \cdot e_1 + \cdots + c(o_{\lambda_n}) \cdot e_n$ 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 $UB^{\mathcal{M}^{\mathcal{I}}}_P(\overline{x}) \mid_{\mathcal{N}}$ 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

$$UB^{\mathcal{M}^I}_{main}(n, m) = c(o_1) \cdot 18 + c(o_{13}) \cdot 6 + c(o_1) \cdot 12 \cdot \mathsf{nat}(n) + c(o_{12}) \cdot 6 \cdot \mathsf{nat}(n) + c(o_{12}) \cdot 8 \cdot \mathsf{nat}(n) \cdot \mathsf{nat}(m) + c(o_{13}) \cdot 2 \cdot \mathsf{nat}(n) \cdot \mathsf{nat}(m) + c(o_{34}) \cdot \mathsf{nat}(n) \cdot \mathsf{nat}(m),$$

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:

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

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 $UB^{\mathcal{M}}_P$ 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 $t \equiv S_0 \rightsquigarrow \cdots \rightsquigarrow S_n$ 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*

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

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

Author's personal copy

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:

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

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 $cobox\_roots(t) = \{\!|\ o_l \mid \_ \rightsquigarrow^{q:\mathsf{newcog}\_}_{o_{j\ldots p}} \_ \in steps(t) \wedge\ l = j \ldots pq\ |\!\}$. Also, given a cobox root o_l, the multiset of objects it owns in a trace t is defined as $obj\_in\_cobox(o_l, t) = \{\!|\ o_{j\ldots pq} \mid \_ \rightsquigarrow^{q:\mathsf{new}\ \_}_{o_{j\ldots p}} \_ \in steps(t) \wedge root(o_{j\ldots p}) = o_l\ |\!\}$.

Example 4.1 Deliberately, the running example shown in Fig. 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 Fig. 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:

Author's personal copy

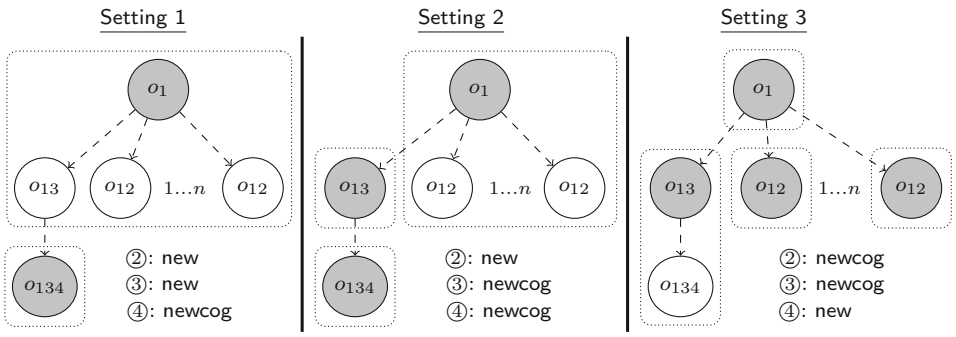


Fig. 6. Configurations for the running example for all settings

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

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

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

■

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:

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

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.

Author's personal copy

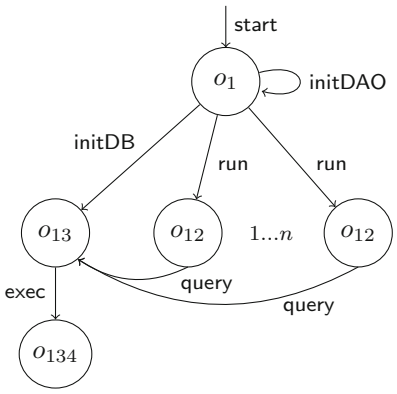


Fig. 7. Interactions for the running example

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

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

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

■

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

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

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

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 Sect. 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.

Author's personal copy

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 $E = \{o_\lambda \rightarrow o_{\lambda'} \mid q{:}y{=}\mathsf{new}\ \_ \text{ or } q{:}y{=}\mathsf{newcog}\ \_ \in P \wedge o_\lambda \in pt(q, this) \wedge o_{\lambda'} \in pt(q, y)\}$.

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 $is\_root(o_{ab\ldots q})$, 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 Fig. 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:

Setting 1: $\langle o_1, \{o_{12}, o_{13}\}\rangle, \langle o_{34}, \{\}\rangle$,
Setting 2: $\langle o_1, \{o_{12}\}\rangle, \langle o_{13}, \{\}\rangle, \langle o_{34}, \{\}\rangle$,
Setting 3: $\langle o_1, \{\}\rangle, \langle o_{12}, \{\}\rangle, \langle o_{13}, \{o_{34}\}\rangle$.

■

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

Note that, in the above definition, function $obj\_names\_in\_cobox(o_\lambda, G_P)$ 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 Fig. 5, we show the points-to graph obtained for method main2 (left of Fig. 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 $covers(\mathcal{N}, O)$ if $\forall\, o_l \in O, \exists\ o_\lambda \in \mathcal{N}$ s.t. $o_\lambda = name(o_l)$. Given $\langle o_l, O\rangle$ and $\langle o_\lambda, \mathcal{N}\rangle$, we write $covers(\langle o_\lambda, \mathcal{N}\rangle, \langle o_l, O\rangle)$ if $o_\lambda = name(o_l)$ and $covers(\mathcal{N}, O)$. Given a trace t, its concrete configuration C_t, and an abstract configuration $\mathcal{A}$, we write $covers(\mathcal{A}, C_t)$ if $\forall \langle o_l, O\rangle \in C_t$ there exists $\langle o_\lambda, \mathcal{N}\rangle \in \mathcal{A}$ s.t. $covers(\langle o_\lambda, \mathcal{N}\rangle, \langle o_l, O\rangle)$.

Author's personal copy

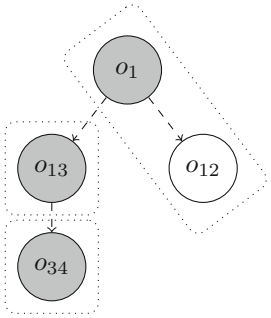


Fig. 8. Points-to graph and abstract configuration for the running example with setting 2

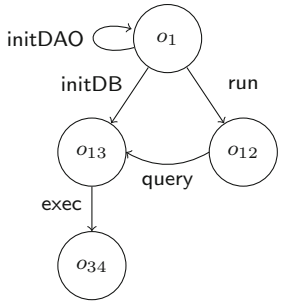


Fig. 9. Interactions graph for the running example

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

Proof. (*Sketch*) The proof relies on the fact that the underlying points-to analysis of Sect. 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 $\langle o_l, obj\_in\_cobox(o_l, t)\rangle$, where l is an allocation sequence for a cobox root (created with newcog). Let $obj\_in\_cobox(o_l, t)$ be the multiset of objects (transitively) created with new from o_l. Let $\mathcal{A}_p$ be a set of pairs $\langle o_\lambda, obj\_names\_in\_cobox(o_\lambda, G_p)\rangle$, where o_λ is an object name for a cobox root. Finally, let $obj\_names\_in\_cobox(o_\lambda, G_p)$ 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 $covers(\langle o_\lambda, \mathcal{N}\rangle, \langle o_l, O\rangle)$ 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 $o_\lambda{=}name(o_l)$ in $\mathcal{O}$. Analogously, for any trace t and object o_l in t such that $o_l{=}root(o_l)$, we have that $covers(obj\_names\_in\_cobox(o_\lambda, G_p), obj\_in\_cobox(o_l, t))$ holds, since all allocation sequences in $obj\_in\_cobox(o_l, t)$ have their corresponding object names in $\mathcal{O}$. Let $o_{l'}$ be an element of $obj\_in\_cobox(o_l, t)$. 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 $o_{\lambda'}{=}name(o_{l'})$ coincides with its allocation sequence: $l' = \lambda'$.
- Otherwise $|l'| > k$, and then λ' in $o_{\lambda'}{=}name(o_{l'})$ 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 $o_{\lambda'} \in obj\_names\_in\_cobox(o_\lambda, G_p)$. □

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 Sect. 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 $cobox(o_\lambda, G_P)$ to denote $\{o_\lambda\} \cup obj\_names\_in\_cobox(o_\lambda, G_P)$.

Author's personal copy

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 $UB^{\mathcal{M}^I}_{main}(n, m) \mid_{cobox(c, G_{main})}$ where c corresponds, in each case, to the cobox name in the considered abstract configuration:

Setting 1		Setting 2		Setting 3	
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 $o_\lambda \in pt(q, this)$ 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 Sect. 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 $UB^{\mathcal{M}^C}_{main}(n, m) \mid_{\{cobox\}}$ 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 $t \equiv S_0 \rightsquigarrow \cdots \rightsquigarrow S_n$ 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*

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

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

Author's personal copy

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

$$\forall\, \overline{x},\ \forall\, t \in executions(P(\overline{x})),\ cost_P(t,\, o_l,\, \mathcal{M}^C) \leq UB_P^{\mathcal{M}^C}(\overline{x})\mid_{\{name(o_l)\}}.$$

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

$$\max_{t \in executions(P(\overline{x}))}(cost_P(t,\, o_l,\, \mathcal{M}^C)) \leq UB_P^{\mathcal{M}^C}(\overline{x})\mid_{\{name(o_l)\}}. \tag{1}$$

(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

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

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

$$card(o_l,\, I) \leq card(name(o_l),\, name(I)).$$

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

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

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 $E = \{o_\lambda \xrightarrow{m} o_{\lambda'} \mid q{:}x!m(\_) \wedge o_\lambda \in pt(q, this) \wedge o_{\lambda'} \in pt(q, x)\}$.

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 $o_\lambda \in pt(q, this)$ 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).

Author's personal copy

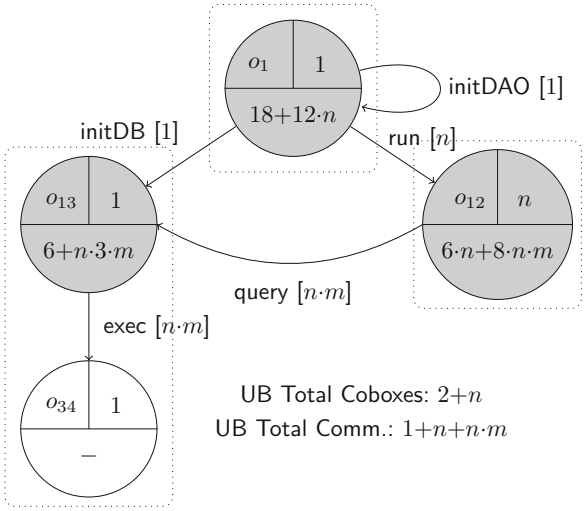


Fig. 10. QA for the running example for setting 3

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.

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

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

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

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

The last row shows the total number of interactions between coboxes. Clearly, the minimum number of inter-cobox interactions happens in setting 1, where most 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 Fig. 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 Sect. 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

Author's personal copy

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 $UB_P^{\mathcal{M}^K}(\overline{x})\mid_{N,M}$, 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 Sect. 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 $t \equiv S_0 \rightsquigarrow \cdots \rightsquigarrow S_n$ 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*

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

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

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

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

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

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

(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 $I_t = \{\!|\ \langle o_l, o_{l'}, m\rangle \mid S \rightsquigarrow_{o_l}^{q:x!m(\_)} \_ \in steps(t) \wedge ob(o_l, \_, \langle tv, \_\rangle, \_) \in S \wedge o_{l'} = tv(x)\ |\!\}$.
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 $cost_P(t, (o_l, o_{l'}), m, \mathcal{M}^K) = \sum_{s \in steps(t)} \mathcal{M}^K(s)\mid_{\{(name(o_l),name(o_{l'}))\},\{m\}}$ (see Sect. 3.2), where $\mathcal{M}^K(s)\mid_{\{(name(o_l),name(o_{l'}))\},\{m\}}$ 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,

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

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

Author's personal copy

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 Sect. 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 Sect. 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, Sect. 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 $\mathcal{M}^C$ and the set of objects $\mathcal{N}$ are the coboxes, i.e., $\mathcal{N} = \{o_1, o_{12}, o_{13}\}$. In particular, $UB^{\mathcal{M}^C}_{main}(n, m) \mid_{\{o_1, o_{12}, o_{13}\}} \leq 32$. Settings 1 and 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 $UB^{\mathcal{M}^C}_P(\overline{x}) \mid_{\{o_1, o_{12}, o_{13}\}}$ 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 $\mathcal{L}_{\langle M, N \rangle}$ for a cost model $\mathcal{M}$ and set of object names $\mathcal{N}$, we say that P conditionally satisfies $\mathcal{L}_{\langle M, N \rangle}$, if there exist conditions φ on the input arguments $\overline{x}$ such that $\varphi \models UB^{\mathcal{M}}_P(\overline{x}) \mid_{\mathcal{N}} \leq \mathcal{L}_{\langle M, N \rangle}$.

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:

Author's personal copy

$$\varphi_{o_1} \equiv 18 + 12 \cdot n \leq 500$$
$$\varphi_{o_{12}} \equiv 6 \cdot n + 8 \cdot n \cdot m \leq 500$$
$$\varphi_{o_{13}} \equiv 6 + 3 \cdot n \cdot m \leq 500.$$

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 Sect. 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:

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

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:

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

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 Sect. 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.

Author's personal copy

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:

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

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:

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

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
$1 - \frac{n \cdot m}{1 + n + 2 \cdot n \cdot m}$	$1 - \frac{n \cdot m + n \cdot m}{1 + n + 2 \cdot n \cdot m}$	$1 - \frac{1 + n + n \cdot m}{1 + n + 2 \cdot n \cdot m}$

■

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 $\mid x^+ - x^- \mid /2$, 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:

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

Author's personal copy

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

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

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 $coboxes(\mathcal{O})$). Then, the mean is captured by the expression:

$$\frac{UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})\mid_{\mathcal{O}}}{UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})\mid_{coboxes(\mathcal{O})}}.$$

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, $UB_P^{\mathcal{M}^{\mathcal{I}}}(\overline{x})\mid_{\{o_\lambda\}}$, by the number of instances of the corresponding cobox $UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})\mid_{\{o_\lambda\}}$. 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, $UB_P^{\mathcal{M}^{\mathcal{C}}}(\overline{x})\mid_{\{o_\lambda\}}$. 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:

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

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:

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

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

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

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

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

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

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

■

Author's personal copy

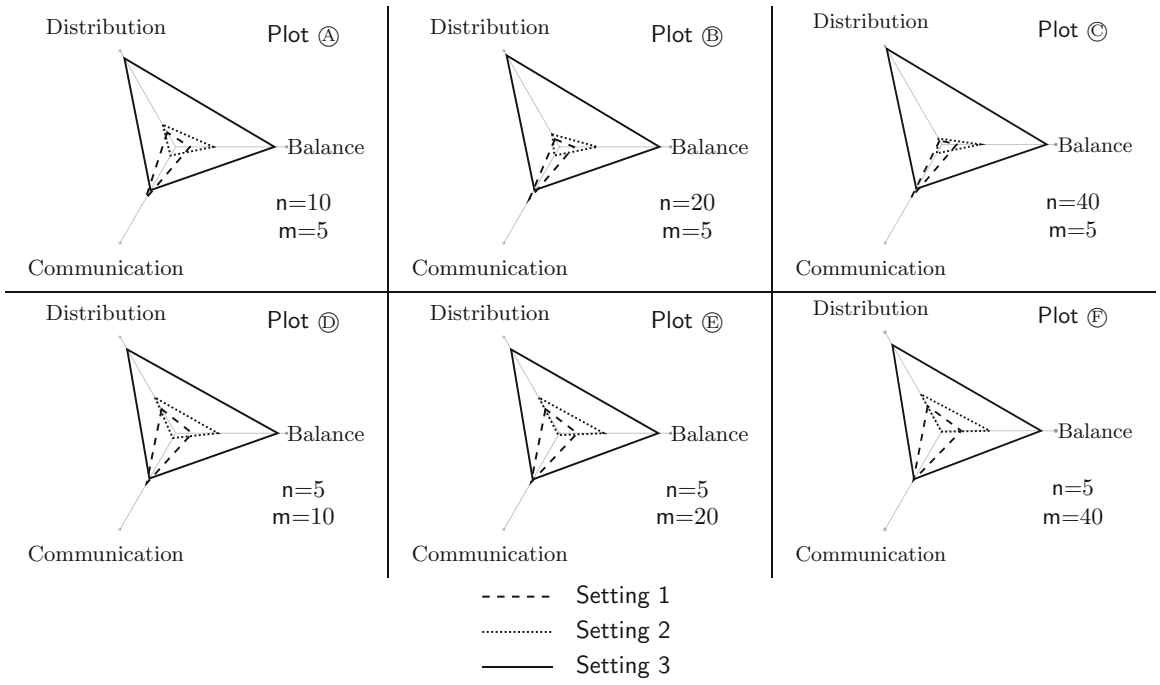


Fig. 11. Comparison for the runnnig example for different values of n and m

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., 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 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 settings 1, 2 and 3, shown in detail in Example 4.1. Such plots will be useful for comparing the behaviour of

Author's personal copy

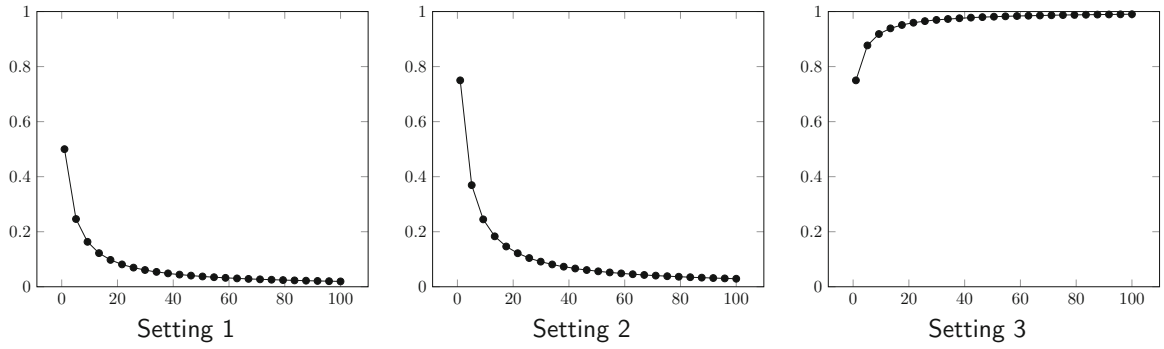


Fig. 12. Graphical representation of the distribution function for the running example

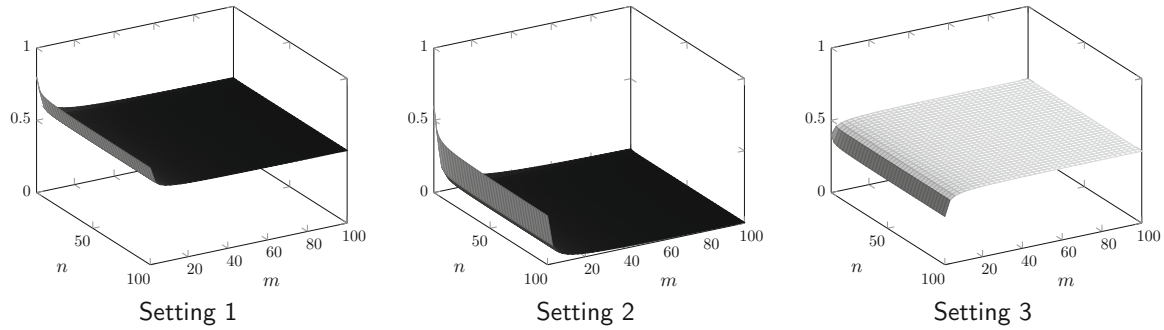


Fig. 13. Graphical representation of the communication function for the running example

different settings for the given input arguments. Let us start by focusing on the plots that correspond to the pairs $\mathsf{n} = 20, \mathsf{m} = 5$ and $\mathsf{n} = 5, \mathsf{m} = 20$. 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 settings 1 and 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, $\mathsf{n} = 20, \mathsf{m} = 5$ and $\mathsf{n} = 5, \mathsf{m} = 20$, setting 3 is the one that best fits with the deployment estimations.

As general conclusion, we can say that settings 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 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 settings 1 and 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.

Author's personal copy

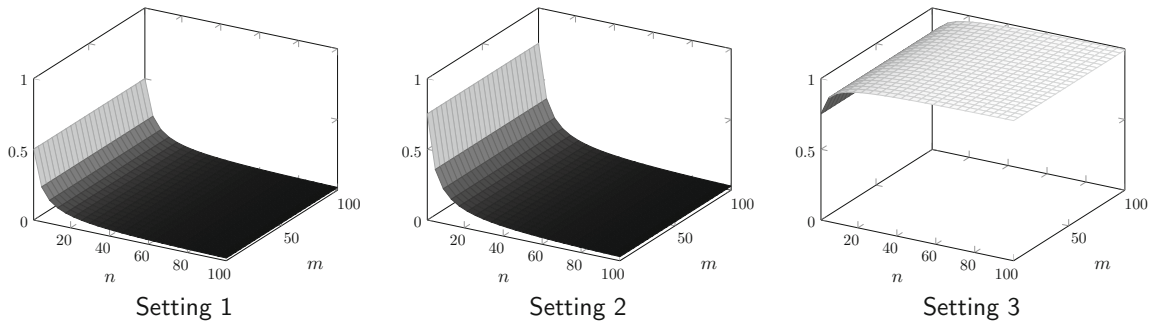


Fig. 14. Graphical representation of the balance function for the running example

In Fig. 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 Fig. 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 Sect. 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 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.

Author's personal copy

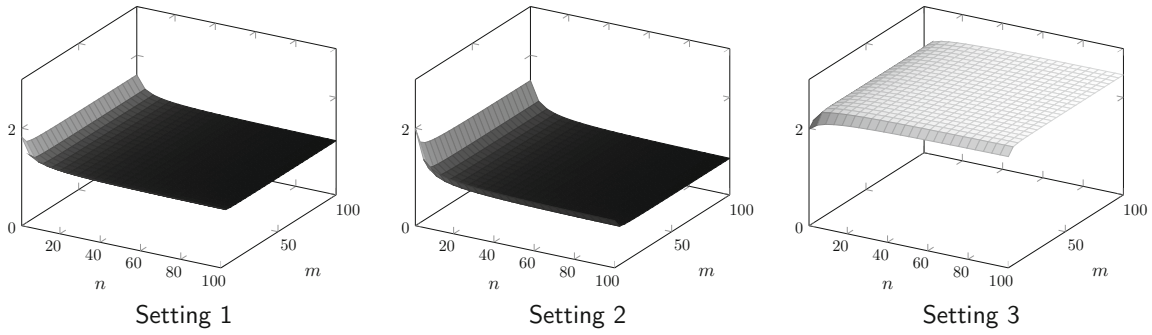


Fig. 15. Graphical representation of the target function $\mathcal{T}_{main}(n, m)$ for the running example

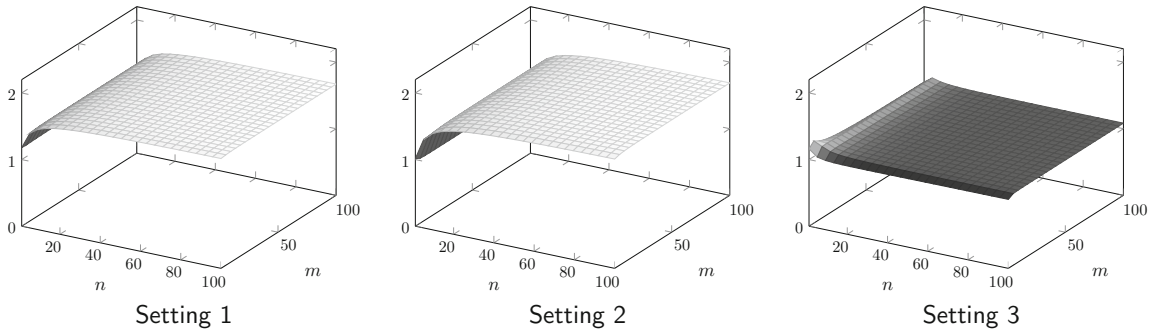


Fig. 16. Graphical representation of the target function $\mathcal{T}'_{main}(n, m)$ for the running example

Example 6.12 Let us start the example with a target functions that weights equally the performance indicators obtained in Examples 6.5, 6.7 and 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 settings 1 and 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 Fig. 16 indicates that the performance obtained for settings 1 and 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 $\mathcal{T}'_{main}(n, m)$. ■

7. Experimental evaluation

We have implemented our analysis in SACO http://costa.ls.fi.upm.es/SACO, a SACO, 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 Fig. 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

Author's personal copy

Table 1. Experimental evaluation results (times in s)

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	2,642/225/1,100
TradingSystem	1,340	15	23	110.31	721.34	746.15	27/2/23	354/9/325	560,390/18,190/309,705

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 3.4 GHz with 4 GB 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 Sect. 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 (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 Sect. 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 Sect. 6.3).

Author's personal copy

Table 2. Experimental evaluation results (times in s)

Benchmark	$\#_{\mathsf{S}}$	$\#_{\mathsf{CP}}$	$\mathbf{T}_{\mathsf{ES}}$	$\#_{\mathsf{CP/s}}$	$\#_{\mathsf{E}}$	$\mathbf{T}_{\mathsf{T}}$
Running	16	100,200	17.48	5,732.26	1,603,200	1.687
BoundedBuffer	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

(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 $\#_{\mathsf{CP/s}}$ 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 12,000 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 ($\#_{\mathsf{CP/s}}$) directly depends on the complexity of the UB expressions and the evaluation significantly varies for different example benchmarks, ranging from 4,347 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 s. In Sect. 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

Author's personal copy

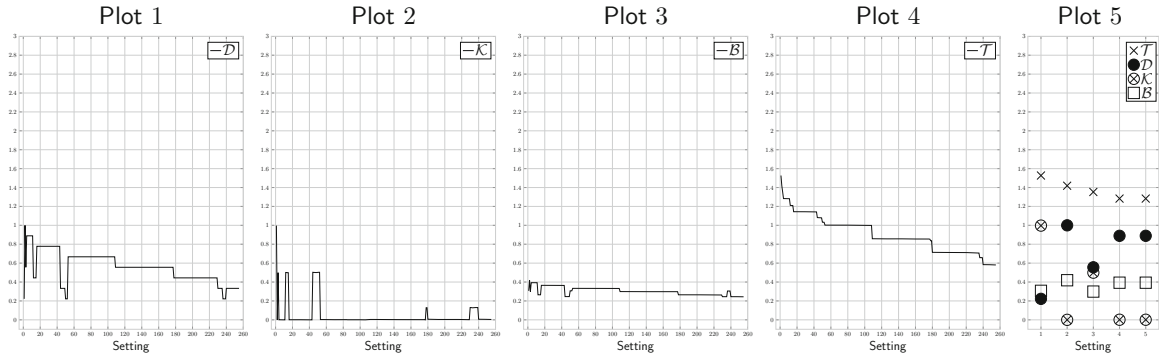


Fig. 17. PeerToPeer case study evaluation (256 settings sorted by $\mathcal{T}$)

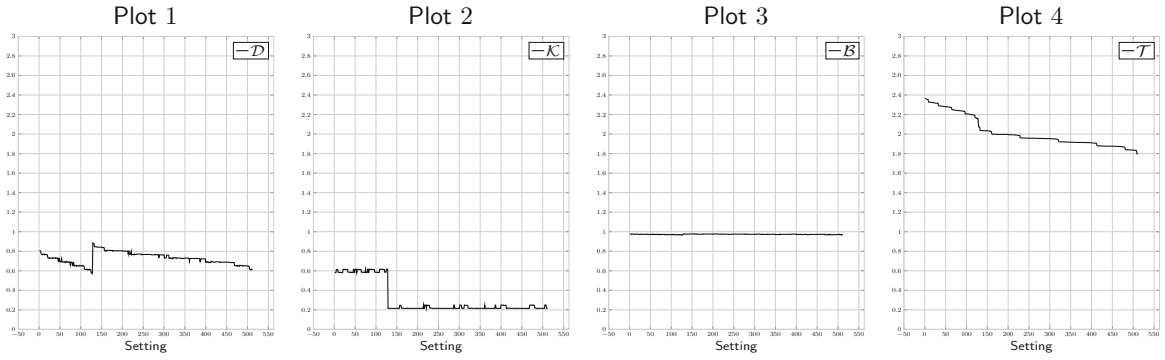


Fig. 18. TradingSystem case study evaluation (512 settings sorted by $\mathcal{T}$)

the 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 Fig. 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 Fig. 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.

Author's personal copy

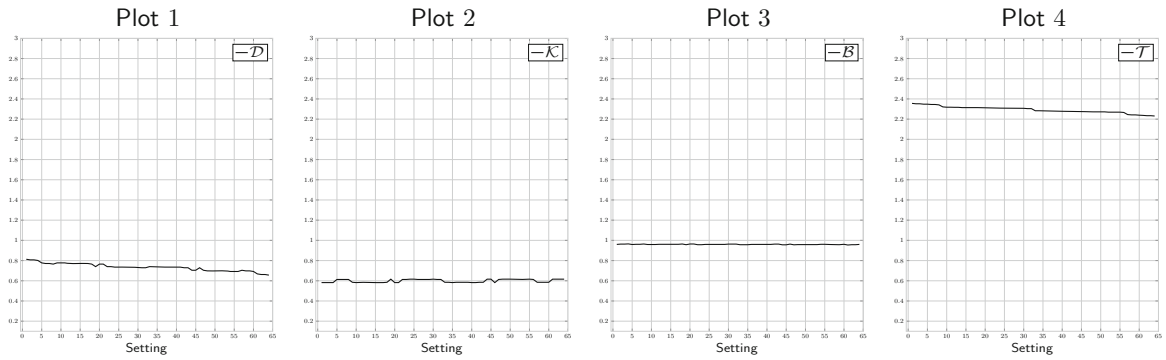


Fig. 19. TradingSystem case study evaluation (64 settings sorted by $\mathcal{T}$)

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 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 Fig. 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 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 Fig. 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 Fig. 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

Author's personal copy

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, GL05]. 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 Kernel Language for Agents Interaction and Mobility (KLAIM) 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

Author's personal copy

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.

Acknowledgments

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.

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 $\mid ab\ldots cd \mid\leq k$, 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 $\mathcal{M}$
$UB_P^{\mathcal{M}}(\overline{x})\mid_{\mathcal{N}}$	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)

Author's personal copy

$Comm_P(\overline{x})$	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)
$\mathcal{M}^K(b)$	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

References

[AAG07] Albert E, Arenas P, Genaim S, Puebla G, Zanardini D (2007) Cost analysis of Java bytecode. In: De Nicola R (ed) Proceedings of the 16th European symposium on programming (ESOP'07). Lecture notes in computer science, vol 4421. Springer, New York, pp 157–172

[AAG10] Albert E, Arenas P, Genaim S, Herraiz I, Puebla G (2010) Comparing cost functions in resource analysis. In: 1st International workshop on foundational and practical aspects of resource analysis (FOPARA'09). Lecture notes in computer science, vol 6234. Springer, New York, pp 1–17

[AAG11a] Albert E, Arenas P, Genaim S, Gómez-Zamalloa M, Puebla G (2011) Cost analysis of concurrent oo programs. In: Proceedings of APLAS'11. LNCS, vol 7078. Springer, New York, pp 238–254

[AAG12a] Albert E, Arenas P, Genaim S, Gómez-Zamalloa M, Puebla G (2012) COSTABS: a cost and termination analyzer for ABS. In: Proceedings of PEPM'12. ACM Press, New York, pp 151–154

[AAG12b] Albert E, Arenas P, Genaim S, Puebla G, Zanardini D (2012) Cost analysis of object-oriented bytecode programs. Theor Comput Sci (Spec Issue Quant Asp Program Lang) 413(1):142–159

[AAG08] Albert E, Arenas P, Genaim S, Puebla G (2008) Automatic inference of upper bounds for recurrence relations in cost analysis. In: Proceedings of SAS'08. Lecture notes in computer science, vol 5079. Springer, New York, pp 221–237

[AAG11b] Albert E, Arenas P, Genaim S, Puebla G (2011) Closed-form upper bounds in static cost analysis. J Autom Reason 46(2):161–203

[ACP13] Albert E, Correas J, Puebla G, Román-Díez G (2013) Quantified abstractions of distributed systems. In: Proceedings of iFM'13. LNCS, vol 7940. Springer, New York, pp 285–300

[AFG13] Albert E, Flores-Montoya A, Genaim S, Martin-Martin E (2013) Termination and cost analysis of loops with concurrent interleavings. In: ATVA'13, LNCS, vol 8172. Springer, New York, pp 349–364

[Ame89] America P (1989) Issues in the design of a parallel object-oriented language. Form Asp Comput 1:366–411

[APC12] Albert E, Arenas P, Correas J, Gómez-Zamalloa M, Genaim S, Puebla G, Román-Díez G (2012) Object-sensitive cost analysis for concurrent objects. http://costa.ls.fi.upm.es/papers/costa/AlbertACGGPRtr.pdf. Accessed 15 May 2014

[ASY09] Al-Saleh MF, Yousif AE (2009) Properties of the standard deviation that are rarely mentioned in classrooms. Austrian J Stat 38(3):193–202

[AVW96] Armstrong J, Virding R, Wistrom C, Williams M (1996) Concurrent programming in Erlang. Prentice Hall, USA

[BCG07] Bruynooghe M, Codish M, Gallagher J, Genaim S, Vanhoof W (2007) Termination analysis of logic programs through combination of type-based norms. TOPLAS 29(2):10

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

[BYV09] Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25(6):599–616

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

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

[PHW05] Di Pierro A, Hankin C, Wiklicky H (2005) Quantitative static analysis of distributed systems. J Funct Program 1:37–81

[Fer01] Feret J (2001) Occurrence counting analysis for the pi-calculus. Electron Notes Theor Comput Sci 39(2):1–18

[FMA13] Flores-Montoya A, Albert E, Genaim S (2013) May-happen-in-parallel based deadlock analysis for concurrent objects. In: FORTE'13. LNCS. Springer, New York, pp 273–288

Author's personal copy

[GL05] Gori R, Levi F (2005) A new occurrence counting analysis for bioambients. In: APLAS. LNCS, vol 3780. Springer, New York, pp 381–400

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

[JHS12] Johnsen EB, Hähnle R, Schäfer J, Schlatte R, Steffen M (2012) ABS: a core language for abstract behavioral specification. In: Proceedings of FMCO'10 (revised papers). LNCS, vol 6957. Springer, New York, pp 142–164

[MRR05] Milanova A, Rountev A, Ryder BG (2005) Parameterized object sensitivity for points-to analysis for Java. ACM Trans Softw Eng Methodol 14:1–41

[SBL11] Smaragdakis Y, Bravenboer M, Lhoták O (2011) Pick your contexts well: understanding object-sensitivity. In: Proceedings of POPL'11. ACM, New York, pp 17–30

[SPH10] Schäfer J, Poetzsch-Heffter A (2010) JCobox: generalizing active objects to concurrent components. In: Proceedings of ECOOP'10. LNCS. Springer, New York, pp 275–299

[YBS86] Yonezawa A, Briot JP, Shibayama E (1986) Object-oriented concurrent programming ABCL/1. In: Proceedings of OOPLSA'86. ACM, New York, pp 258–268

Received 15 November 2013

Revised 31 May 2014

Accepted 12 September 2014 by Einar Broch Johnsen, Luigia Petre, and Michael Butler

Appendix E

Article *Static Inference of Transmission Data Sizes in Distributed Systems*, [8]

Static Inference of Transmission Data Sizes in Distributed Systems

Elvira Albert[1], Jesús Correas[1], Enrique Martin-Martin[1], Guillermo Román-Díez[2]

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

Abstract. We present a static analysis to infer the amount of data that a distributed system may transmit. The different locations of a distributed system communicate and coordinate their actions by posting tasks among them. A task is posted by building a message with the task name and the data on which such task has to be executed. When the task completes, the result can be retrieved by means of another message from which the result of the computation can be obtained. Thus, the transmission data size of a distributed system mainly depends on the amount of messages posted among the locations of the system, and the sizes of the data transferred in the messages. Our static analysis has two main parts: (1) we over-approximate the sizes of the data at the program points where tasks are spawned and where the results are received, and (2) we over-approximate the total number of messages. Knowledge of the transmission data sizes is essential, among other things, to predict the bandwidth required to achieve a certain response time, or conversely, to estimate the response time for a given bandwidth. A prototype implementation in the SACO system demonstrates the accuracy and feasibility of the proposed analysis.

1 Introduction

Distributed systems are increasingly used in industrial processes and products, such as manufacturing plants, aircraft and vehicles. For example, many control systems are decentralized using a distributed architecture with different processing locations interconnected through buses or networks. The software in these systems typically consists of concurrent tasks which are statically allocated to specific locations for processing, and which exchange messages with other tasks at the same or at other locations to perform a collaborative work. A decentralized approach is often superior to traditional centralized control systems in performance, capability and robustness. Systems such as control systems are often critical: they have strict requirements with respect to timing, performance, and stability. A failure to meet these requirements may have catastrophic consequences. To verify that a given system is able to provide the required quality of control, an essential aspect is to accurately predict the communication traffic among its distributed components, i.e., the amount of data to be transmitted along any execution of the distributed system.

In order to estimate the transmission data sizes, we need to keep track of the amount of data transmitted in two ways: (1) by posting asynchronous tasks among the locations, this requires building a message in which the name of the task to execute and the data on which it executes are included; (2) by retrieving the results of executing the tasks, in our setting, we use future variables [8] to synchronize with the completion of a task and retrieve the result. This paper presents a static analysis to infer a safe over-approximation of the transmission data sizes required by both sources of communications in a distributed system. Our method infers three different pieces of information:

1. *Inference of distributed locations.* As locations can be dynamically created, in a first step, we need to find out the locations that compose the system and give them abstract names which will allow us to track communications among them during the analysis. This is formalized by means of points-to analysis [14,13], a typical analysis in pointer-based languages which infers the memory locations that a reference variable can point to. In our case, locations are referenced from reference variables, thus the use of points-to analysis.
2. *Inference of number of tasks spawned.* The second step is to infer an upper bound on the number of tasks spawned between each pair of distributed locations. This is a problem which can be solved by a generic cost analysis framework such as [3]. In particular, we need to use a *symbolic* cost model which allows us to annotate the caller and callee locations when a task is spawned in the program. In essence, if we find an instruction a!m(x) which spawns a task m at location a, the cost model symbolically counts $c(this, a, m) * 1$, i.e., it counts that 1 task executing m is spawned from the current location this at a. If the task is spawned within a loop that performs n iterations, the analysis will infer $c(this, a, m) * n$.
3. *Inference of data sizes.* Finally, we need to infer the sizes of the arguments in the task invocations. Typically, size analysis [7] infers upper bounds on the data sizes at the end of the program execution. Here, we are interested in inferring the sizes at the points in which tasks are spawned. In particular, given an instruction a!m(x), we aim at over-approximating the size of x when the program reaches the above instruction. If the above instruction can be executed several times, we aim at inferring the largest size of x, denoted $\alpha(x)$, in all executions of the instructions. Altogether, $c(this, a, m) * \alpha(x)$ is a safe over-approximation of the data size transmission due to such instruction. The analysis will infer such information for each pair of locations in the system that communicate, annotating also the task that was spawned.

We demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer within the SACO system [2], a static analyzer for distributed concurrent programs. Preliminary experiments on some typical applications for distributed programs show the feasibility and accuracy of our analysis. The tool can be used on-line from a web interface available at http://costa.ls.fi.upm.es/web/saco.

The remaining of the paper is organized as follows. The next section will present the distribution model that we use to formalize the analysis. Sec. 3 defines the concrete notion of transmission data size that we then want to over-approximate by means of static analysis. Sec. 4 presents the static analysis that carries out the three steps mentioned above. Sec. 5 reports on preliminary experimental results and Sec. 6 concludes.

2 Distribution Model

We consider a distributed programming model with explicit locations and based on the actor-based paradigm [1]. Each location represents a processor with a procedure stack and an unordered queue of pending tasks. Initially all processors are idle. When an idle processor's task queue is not empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the queues of any processor (*message passing*), including its own, and synchronize with the completion of tasks. This synchronization is done by means of *future variables* [8]. When a task completes or when it is awaiting for another task to terminate, its processor becomes idle again, chooses the next pending task, and so on. This distribution model captures the essence of the concurrency model of languages like X10 [12], Erlang [6], Scala [10] or ABS [11].

2.1 Syntax

Regarding data, the language contains basic types B (int, bool ...) and parametric data types D. Data types are declared by listing all the possible constructors C and their arguments, a syntax similar to functional languages like *Haskell*:

(Type variable)	N	$::= a, b, c \ldots$
(Basic type)	B	$::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{void} \mid \ldots$
(Data type declaration)	Dd	$::= \mathsf{data}\ D(N_1, \ldots, N_n) = C_1 \mid \ldots \mid C_k \quad (n \geq 0, k > 0)$
(Constructor)	C	$::= Co(N_1, \ldots, N_n) \quad (n \geq 0)$
(Ground type)	T	$::= B \mid D(T_1, \ldots, T_n) \quad (n \geq 0)$

Example 1 (Data types). We define integer lists and general binary trees as:

$\mathsf{data\ List} = \mathsf{Nil} \mid \mathsf{Cons}(\mathsf{int}, \mathsf{List})$

$\mathsf{data\ Tree}(a) = \mathsf{Leaf}(a) \mid \mathsf{Branch}(a, \mathsf{Tree}(a), \mathsf{Tree}(a))$

Using the previously declared constructors the list $l = [1, 2, 3]$ is defined as $\mathsf{l} = \mathsf{Cons(1,\ Cons(2,\ Cons(3,Nil)))}$, and the binary tree t with 2 at the root, 1 as left child and 3 as right child as $\mathsf{t} = \mathsf{Branch(2,\ Leaf(1),\ Leaf(3))}$

Apart from data type declarations, the language allows the definition of functions based on pattern matching as in functional languages—e.g. head, tail, length, etc. This syntax has been omitted for the sake of conciseness, as it does not play an important role for presenting the analysis.

Regarding programs, the number of distributed locations needs not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore

```
1 main (List l, int s) {
2   x = newLoc;
3   y = newLoc;
4   z = newLoc;
5   x!extend(l,s);
6 }
7
8 int foo (int i) {
9   return i;
10 }
```

```
11 void extend (List l,int s) {
12   while(s > 0) {
13     Fut f= y!add(l,5);
14     await f?;
15     l = f!get;
16     z!process(l);
17     s = s − 1;
18   }
19 }
```

```
20 List add (List l, int e) {
21   List r = Cons(e,l);
22   return r;
23 }
24 void process (List le) {
25   while(le != Nil) {
26     Int h = head(le)
27     y!foo(h);
28     le = tail (le);
29   }
30 }
```

Fig. 1. Running Example

be similar to an *object* and can be dynamically created using the instruction newLoc. The program is composed by a set of methods finished with a return instruction $M ::= T\ m(\bar{T}\ \bar{x})\{s; \mathsf{return}\ x;\}$ where s takes the form:

$$s ::= s; s \mid x = e \mid x = f.\mathsf{get} \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid b = \mathsf{newLoc} \mid f = b!m(\bar{x}) \mid \mathsf{await}\ f?$$

The notation $\bar{T}$ is used as a shorthand for $T_1, \ldots, T_n$, and similarly for other names. The special location identifier *this* denotes the current location. For the sake of generality, the syntax of expressions e is left open. The semantics of future variables f and concurrency instructions is explained below.

Example 2 (running example). Fig. 1 shows a method main which creates three distributed locations, x, y and z, and receives a list of integers, l, and one integer, s. In the example, we assume that x, y and z are global variables and thus accessible to all methods. Also, we have omitted return instructions in void tasks. Method main spawns task extend at location x in Line 5 (L5 for short) and sends data l and x (thus there is data transmission at this point). Method extend extends l with s new elements. To do this, it invokes method add at location y that extends the list with a new element (L13). The await instruction at L14 awaits for the termination of add. The result is retrieved using the get instruction at L15, where besides we assign the result to l. Within the loop of extend, tasks executing process are spawned at location z. The execution of process traverses the list in the while loop and invokes foo for each element in l. An important point to note is that, besides the data transmitted when asynchronous tasks are spawned, the instruction get also involves data transmission to retrieve the results.

2.2 Semantics

A *program state* has the form $loc_1 \| \ldots \| loc_n$, denoting the currently existing distributed locations. Each *location* is a term $loc(lid, tid, \mathcal{Q})$ where *lid* is the location identifier, *tid* is the identifier of the *active task* which holds the location's

$$\frac{t = tsk(tid, m, l, \langle x = \mathsf{newLoc}; s\rangle),\ \mathrm{fresh}(lid_1)\ ,\ l' = l[x \to lid_1]}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \bot, \{\})} \quad (\textsc{newLoc})$$

$$\frac{l(x) = lid_1,\ \mathrm{fresh}(tid_1),\ l_1{=}buildLocals(\bar{z}, m_1),\ l' = l[f \to \langle tid_1, \bot, \bot\rangle]}{loc(lid, tid, \{tsk(tid, m, l, \langle f{=}x!m_1(\bar{z}); s\rangle)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \mathcal{Q}') \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1)) \cup \mathcal{Q}'\})} \quad (\textsc{async})$$

$$\frac{l(x) = v, l_1(f) = \langle tid, \bot, \bot\rangle, l_1' = l_1[f \to \langle tid, true, \bot\rangle]}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{return}\ x\rangle)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1, \_)\} \cup \mathcal{Q}_1) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \epsilon(v))\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1', \_)\} \cup \mathcal{Q}_1)} \quad (\textsc{return})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle), l(f) = \langle tid_1, true, \_\rangle}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, s)\} \cup \mathcal{Q})} \quad (\textsc{await-t})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle), l(f) = \langle tid_1, \bot, \bot\rangle}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q})} \quad (\textsc{await-f})$$

$$\frac{l(f) = \langle tid_1, true, \bot\rangle, l' = l[x \to v, f \to \langle tid_1, true, v\rangle]}{loc(lid, tid, \{tsk(tid, m, l, \langle x = f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1, \epsilon(v))\} \cup \mathcal{Q}_1) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, \_, l_1, \epsilon(v))\} \cup \mathcal{Q}_1)} \quad (\textsc{get-r})$$

$$\frac{l(f) = \langle tid_1, true, v\rangle, v \neq \bot, l' = l[x \to v]}{loc(lid, tid, \{tsk(tid, m, l, \langle x = f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q})} \quad (\textsc{get-l})$$

$$\frac{select(\mathcal{Q}) = tid,\ t = tsk(tid, \_, \_, s){\in}\mathcal{Q}, s \neq \epsilon(v)}{loc(lid, \bot, \mathcal{Q}){\rightsquigarrow}loc(lid, tid, \mathcal{Q})} \quad (\textsc{select})$$

Fig. 2. (Summarized) Semantics for Distributed Execution

lock or $\bot$ if the lock is free, and $\mathcal{Q}$ is the set of tasks at the location. Only one task, which holds the location's *lock*, can be *active* (running) at this location. All other tasks are *pending*, waiting to be executed, or *finished*, if they terminated and released the lock. A *task* is a term $tsk(tid, m, l, s)$ where tid is a unique task identifier, m is the name of the method executing in the task, l is a mapping from local variables to their values and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated with value v.

The execution of a program starts from a method m in an initial state S_0 with a single (initial) location with identifier 0 executing task 0 of the form $S_0{=}loc(0, 0, \{tsk(0, \mathsf{m}, l, body(\mathsf{m}))\})$. Here, l maps parameters to their initial val-

ues and local references to null (standard initialization), and $body(\mathsf{m})$ refers to the sequence of instructions in the method m. The execution proceeds from the initial state S_0 by selecting *non-deterministically* one of the locations and applying the semantic rules depicted in Fig. 2. The treatment of sequential instructions is standard and thus omitted. The operational semantics $\leadsto$ is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows. In NEWLOC, the active task *tid* at location *lid* creates a location lid_1 which is introduced to the state with a free lock. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is added to the queue of location lid_1—the case $lid{=}lid_1$ is analogous, the new task tid_1 is simply added to the queue $\mathcal{Q}$ of *lid*. The future variable f allows synchronizing the execution of the current task with the completion of the created task, and retrieving its result. The association of the future variable to the task is stored in the local variables table $l'(f){=}\langle tid_1, \bot, \bot\rangle$: the future variable f is linked to task tid_1, the task has not terminated yet (first $\bot$ in the tuple), and the result of the invocation is not available yet (second $\bot$). The rule RETURN is used when a task *tid* executes a return instruction. The terminating task *tid* finishes the execution with value v (its sequence of instructions is set to $\epsilon(v)$) and the calling task tid_1 is notified that *tid* has terminated by setting to *true* the termination flag of the corresponding future variable—the case $lid{=}lid_1$ is analogous, but storing the termination flag in a task in queue $\mathcal{Q}$. In AWAIT-T, the future variable we are awaiting for points to a finished task (it has the termination flag set to *true* in the future variable f stored in the local variable table l) and await can be completed. Otherwise, AWAIT-F yields the lock so that any other task of the same location can take it. The rule GET-R retrieves the returning value from the task tid_1 linked to the future variable f, if the corresponding task has terminated and the value has not been retrieved before. If tid_1 has not terminated, it will wait for the value without yielding the lock. If the returning value has been retrieved from the remote object already, it is copied locally from the future variable f by means of GET-L. Finally, in rule SELECT an idle location takes a non-finished task to continue the execution—the function $select(\mathcal{Q})$ non-deterministically returns a task identifier occurring in $\mathcal{Q}$.

Example 3 (semantics). The following sequence is the beginning of a trace of the program in Fig. 1 starting from main(Cons(1,Cons(2,Nil)),7). For the sake of conciseness we represent lists with square brackets—[1,2]—instead of constructors and we use l_e, l_a and l_p to denote initial local mappings, stressing only the important changes to them at each step.

$$
\begin{aligned}
S_0 &\equiv loc(0,0,\{tsk(0,\mathsf{main},l_m,\langle \mathsf{x = newLoc;}\ldots\rangle)\}) \leadsto^{\text{NEWLOC}\times 3}\\
S_3 &\equiv loc(0,0,\{tsk(0,\mathsf{main},l_m[\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 3],\langle \mathsf{x!extend(l,s)}\rangle)\}) \parallel loc(1,\bot,\{\})\\
&\quad \parallel loc(2,\bot,\{\}) \parallel loc(3,\bot,\{\}) \leadsto^{\text{ASYNC}}\\
S_4 &\equiv loc(0,0,\ldots) \parallel loc(1,\bot,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{while}\ (s>0)\{\ldots\}\rangle)\})\\
&\quad \parallel loc(2,\bot,\{\}) \parallel loc(3,\bot,\{\}) \leadsto^{\text{SELECT}} S_5 \leadsto\\
S_6 &\equiv loc(0,0,\ldots) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{Fut\ f{=}y!add(l,5);}\ldots\rangle)\})\\
&\quad \parallel loc(2,\bot,\{\}) \parallel loc(3,\bot,\{\}) \leadsto^{\text{ASYNC}}\\
S_7 &\equiv loc(0,0,\ldots) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e[\mathsf{f}\mapsto\langle 2,\bot,\bot\rangle],\langle \mathsf{await\ f?;}\ldots\rangle)\})\\
&\quad \parallel loc(2,\bot,\{tsk(2,\mathsf{add},l_a,\langle \mathsf{List\ r = Cons(e,l);return\ r}\rangle)\}) \parallel loc(3,\bot,\{\}) \leadsto^{\text{SELECT}}
\end{aligned}
$$

$S_8 \equiv loc(0,0,\ldots)) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{await\ f?};\ldots\rangle)\})$
$\quad \parallel loc(2,2,\{tsk(2,\mathsf{add},l_a,\langle \mathsf{List\ r = Cons(e,l);return\ r}\rangle)\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow S_9 \rightsquigarrow^{\text{RETURN}}$
$S_{10} \equiv loc(0,0,\ldots) \parallel loc(1,1,\{tsk(1,\mathsf{extend},l_e[\mathsf{f} \mapsto \langle 2,true,\bot\rangle],\langle \mathsf{await\ f?};\ldots\rangle)\})$
$\quad \parallel loc(2,\bot,\{tsk(2,\mathsf{add},l_a,\epsilon([5,1,2]))\}) \parallel loc(3,\bot,\{\}) \rightsquigarrow^{\text{AWAIT-T+GET-R}}$
$S_{12} \equiv loc(0,0,\ldots) \parallel loc(2,\bot,\{tsk(2,\mathsf{add},l_a,\epsilon([5,1,2]))\}) \parallel loc(3,\bot,\{\}) \parallel loc(1,1,$
$\quad \{tsk(1,\mathsf{extend},l_e[\mathsf{f}\mapsto\langle 2,true,[5,1,2]\rangle,\mathsf{l}\mapsto[5,1,2]],\langle \mathsf{z!process(l)};\ldots\rangle)\}) \rightsquigarrow^{\text{ASYNC}}$
$S_{13} \equiv loc(0,0,\ldots) \parallel loc(2,\bot,\ldots) \parallel loc(3,\bot,\{tsk(3,\bot,l_p,body(\mathsf{process}))\})$
$\quad loc(1,1,\{tsk(1,\mathsf{extend},l_e,\langle \mathsf{s = s - 1};\ldots\rangle)\})$

From state S_0 to S_3 we create the three locations x(1), y(2) and z(3) applying rule NEWLOC. In S_3 a new task extend is spawned using rule ASYNC, that is placed in the queue of location 1. Since location 1 is idle but the queue contains the non-finished task 2 in S_4, it takes the lock (SELECT) and executes the first iteration of the loop. In S_6 and S_7 a new task add is spawned to location 2 and it takes the lock. Note that in S_7 the local mapping is extended to store that the future variable f is linked to task 2, which is not finished yet ($\bot$). Task 2 finishes immediately by assigning variable r and returning: it stores the final value [5,1,2] and notifies task 1 (RETURN). Since task 2 is finished in S_{10} the await and get instructions can proceed (rules AWAIT-T and GET-R resp.), yielding to S_{12}. Finally, task 2 spawns a new task process in location 3.

3 The Notion of Transmission Data Size

The *transmission data size* of a program execution is the total amount of data that is moved between locations. There are two situations that generate data movement between locations: a) when a task is invoked (in this case it sends a message to the destination location containing all the arguments); and b) when the returning value of a task invocation is retrieved (it sends a message containing that value). Therefore, only these two transitions of states will contribute to the transmission data size of a program execution. In order to define this notion we will consider that state transitions are decorated with transmission data size information: $S_1 \rightsquigarrow^{d}_{(lid_1,lid_2,m)} S_2$, meaning a transmission of d units of data from object lid_1 to lid_2 through m. Transitions that do not generate data transmission will be decorated as $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$. Since we are considering an abstract representation of data by means of functional types, we will focus on *units of data* transmitted instead of bits, which depends on the actual implementation and is highly platform-dependent. Concretely, we assume that the cost of transmitting a basic value or a data type constructor is one unit of data. This size measure is known as *term size*. However, the static analysis we propose later would work also with any other mapping from data types to corresponding sizes (given by means of a function α such as the one below).

Definition 1 (term size). *The* term size *of value v—$\alpha(v)$—is defined as:*

$$\alpha(v) = \begin{cases} 1 + \sum_{i=1}^{n} \alpha(v_i) & \text{if } v = Co(v_1 \ldots v_n), \\ 1 & \text{otherwise.} \end{cases}$$

Example 4 (size measures). Considering the term size measure, the size of the list $\mathsf{l} = \mathsf{Cons(1,\ Cons(2,\ Cons(3,Nil)))}$ is $\alpha(\mathsf{l}) = 7$ (4 data constructors and 3 integers) and the size of the tree $\mathsf{t} = \mathsf{Branch(2,\ Leaf(1),\ Leaf(3))}$ is $\alpha(\mathsf{t}) = 6$ (3 constructors plus 3 integers).

Definition 2 (decorated step). *A step $S_1 \rightsquigarrow S_2$ using rule R from Fig. 2 is decorated as follows:*

- *If $R =$ ASYNC then the step is decorated as $S_1 \rightsquigarrow^{d}_{(lid,lid_1,m)} S_2$, where $d = \mathcal{I} + \sum_{z \in \overline{z}} \alpha(l(z))$, and m is the method invoked in the call. The constant $\mathcal{I}$ is the size of establishing the communication, and we add the size of all the arguments passed to the destination location. Note that a task invocation inside the same location ($lid = lid_1$) will not generate any transmission, so in these cases the decoration is $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$.*
- *If $R =$ GET-R then the decorated step is $S_1 \rightsquigarrow^{d}_{(lid_2,lid_1,m)} S_2$, where $d = \mathcal{I} + \alpha(v)$, v corresponds to the returned value, and m is the method that returned v. As before, if $lid = lid_1$ then there is no transmission and the decoration is $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$.*
- *If $R \in \{$NEWLOC, RETURN, AWAIT-T, AWAIT-F, GET-L, SELECT$\}$, then the step does not move any data, so it is decorated with an empty label: $S_1 \rightsquigarrow^{0}_{\epsilon} S_2$.*

Observe that rules AWAIT-T, AWAIT-F and GET-L use local variables only, and therefore do not perform any remote communication. Rule RETURN notifies the termination of a method to the caller location, although its cost is included in the size $\mathcal{I}$ for establishing the communication included in rule ASYNC.

Definition 3 (transmission data size of a trace). *Given a decorated trace $\mathcal{T} \equiv S_0 \rightsquigarrow^{d_1}_{o_1} S_1 \rightsquigarrow^{d_2}_{o_2} \ldots \rightsquigarrow^{d_n}_{o_n} S_n$, the transmission data size of $\mathcal{T}$—trans($\mathcal{T}$)—is defined as:*

$$trans(\mathcal{T}) = \sum_{i=1}^{n} d_i$$

Example 5 (transmission data size). The decorated trace from Ex. 3 is:

$$\mathcal{T}_d \equiv S_0 \rightsquigarrow^{0}_{\epsilon} S_1 \rightsquigarrow^{0}_{\epsilon} S_2 \rightsquigarrow^{0}_{\epsilon} S_3 \rightsquigarrow^{\mathcal{I}+6}_{(0,1,\mathsf{extend})} S_4 \rightsquigarrow^{0}_{\epsilon} S_5 \rightsquigarrow^{0}_{\epsilon} S_6 \rightsquigarrow^{\mathcal{I}+6}_{(1,2,\mathsf{add})} S_7 \rightsquigarrow^{0}_{\epsilon} S_8 \rightsquigarrow^{0}_{\epsilon} S_9 \rightsquigarrow^{0}_{\epsilon} S_{10} \rightsquigarrow^{0}_{\epsilon} S_{11} \rightsquigarrow^{\mathcal{I}+7}_{(2,1,\mathsf{add})} S_{12} \rightsquigarrow^{\mathcal{I}+7}_{(1,3,\mathsf{process})} S_{13}$$

From S_3 to S_4 it sends a message ($\mathcal{I}$) from location 0 to 1 containing the arguments of the call: l=Cons(1,Cons(2,Nil)) and s=7, where $\alpha(l) = 5$ and $\alpha(7) = 1$. Similarly, from S_6 to S_7 it sends a message from location 1 to 2 with the arguments l and 5 for task add. In State S_9 it executes a return instruction, that notifies the termination to the caller, but its size is already considered in the call (S_6). The returning value from the call to add is actually received from the caller at S_{12}, by means of a message from location 2 to 1 with the returning value $\mathsf{r} = \mathsf{Cons(5,Cons(1,Cons(2,Nil)))}$, $\alpha(r) = 7$. Finally, the invocation of task process in state S_{12} sends a message from location 1 to 3 containing the argument $\mathsf{l} =$

Cons(5,Cons(1,Cons(2,Nil))), of size 7. Considering this decorated trace, the total transmission data size is:

$$trans(\mathcal{T}_d) = (\mathcal{I}+6)+(\mathcal{I}+6)+(\mathcal{I}+7)+(\mathcal{I}+7) = 4{*}\mathcal{I}+26$$

In other words, the transmission data size is $4{*}\mathcal{I}$ units of data for creating 4 messages, and 26 units of data for the transmission of values.

The transmission data size of a trace takes into account all the invocation and returning messages, independently of the location involved. In our setting we have several locations that can be executing in different machines or CPUs, so it is interesting to limit transmission data size to some locations. We define a *restriction* operator over traces to consider only data-moving steps between certain locations.

Definition 4 (trace restriction). *Given a decorated trace $\mathcal{T}$, two location identifiers, l_1 and l_2, a method m, the trace restriction $\mathcal{T}|_{l_1 \xrightarrow{m} l_2}$ is defined as:*

$$\mathcal{T}|_{l_1 \xrightarrow{m} l_2} = \{S_{i-1} \rightsquigarrow^{d_i}_{(l_1,l_2,m)} S_i \mid S_{i-1} \rightsquigarrow^{d_i}_{(l_1,l_2,m)} S_i \in \mathcal{T}\}$$

4 Automatic Inference of Transmission Data Sizes

The analysis has three main parts which are introduced in the following sections: Sec. 4.1 is encharged of inferring the locations in the distributed system and using them to define the *cost centers* on which the cost analysis is based; Sec. 4.2 infers upper bounds on the number of tasks spawned along any execution of the program; Sec. 4.3 over-approximates the sizes of the data transmitted when spawning asynchronous calls and when retrieving their results.

4.1 Inference of Distributed Locations

Since locations can be dynamically created, we need an analysis that abstracts them into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. *Points-to* analysis [14,13,15] solves this problem. It infers the set of memory locations that a reference variable can *point-to*. Different abstractions can be used and our method is parametric on the chosen abstraction. Any points-to analysis that provides the following information with more or less accurate precision can be used (our implementation uses [13]): (1) $\mathcal{O}$, the set of abstract locations; (2) a function $pt(pp, v)$ that, for a given program point pp and variable v, returns the set of abstract locations in $\mathcal{O}$ to which v may point.

Example 6 (distributed locations). Consider the main method shown in Fig. 1 which creates three locations x, y and z at L2, L3 and L4, and which are abstracted, respectively, as o_x, o_y and o_z. By using the points-to analysis we obtain the following set of objects created along the execution of main, $\mathcal{O} = \{o_x, o_y, o_z\}$.

Besides, the points-to analysis can infer information for the local variables at the level of program point, that is, $pt(L11, \mathsf{this}) = \{o_x\}$, $pt(L13, \mathsf{y}) = \{o_y\}$, $pt(L16, \mathsf{z}) = \{o_z\}$, $pt(L20, \mathsf{this}) = \{o_y\}$, $pt(L24, \mathsf{this}) = \{o_z\}$, $pt(L26, \mathsf{y}) = \{o_y\}$ or $pt(L8, \mathsf{this}) = \{o_y\}$.

The distributed locations that the points-to analysis infers are used to define the *cost centers* [3] that the resource analysis will use. The notion of cost center is used to attribute the cost of each instruction to the location that executes it. In the above example, we have three locations which lead to three cost centers, $c(o_x)$, $c(o_y)$ and $c(o_z)$.

4.2 Inference of number of tasks spawned

Our analysis builds upon well-established work on cost analysis [9,16,3]. Such analyses are based on a generic notion of resource which can be instantiated to measure different metrics such as number of executed instructions, amount of memory created, number of calls to methods, etc. In particular, the *cost model* is used to determine the type of resource we are measuring. Traditionally, a cost model is a function $\mathcal{M} : \mathit{Instr} \to \mathbb{N}$ which, for each instruction in the program, returns a natural number which represents its cost. As examples of cost models we could have: for counting the number of instructions executed by a program, the cost model counts one unit for any instruction, i.e., $\mathcal{M}^i(ins) = 1$; for counting the number of calls, we can use $\mathcal{M}^c(ins) = 1$ if $ins \equiv x!m(\_)$; and 0 otherwise. When the analysis uses cost centers, the cost model additionally defines to which cost center the cost must be attributed. For instance, when counting number of instructions, we have that $\mathcal{M}(i) = \sum_{o \in pt(pp,this)} c(o) * 1$, where pp is the program point of instruction i, i.e., the instruction is accumulated in all locations that it can be executed (this is given by the locations to which the this reference can point).

In what follows, we use the cost analyzer as a black box in the following way. Given a method $m(\bar{x})$ and a cost model, the cost analyzer gives us an *upper bound* for the total cost (for the resource specified in the cost model) of executing m of the form $\mathcal{U}_m(\bar{x}) = \sum_{i=1}^{n} cc_i * C_i$, where cc_i is a cost center and C_i is a cost expression that bounds the cost of the computation carried out by the cost center cc_i. If one is interested in studying the computation performed by one particular cost center cc_j, we simply replace all cc_i with $i \neq j$ by 0 and cc_j by 1. In order to obtain the cost expression C_i, the cost analyzer needs to over-approximate the number of iterations that loops perform, and infer the maximum sizes of data. For the sake of this paper, we do not need to go into the technical details of this process. To infer an upper bound on the number of tasks spawned by the program, we simply have to define a *number of tasks cost model* and use the cost analyzer as a black box.

Definition 5 (number of tasks cost model). *Given an instruction ins at program point pp, we define the number of tasks cost model, $\mathcal{M}^t(ins)$ as a function which returns $c(o_1, o_2, m)$ if $ins \equiv f{=}y!m(\_) \wedge o_1 \in pt(pp, this) \wedge o_2 \in pt(pp, y) \wedge o_1 \neq o_2$, and 0 otherwise.*

The main feature of the above cost model is that we use an extended form of cost centers which are triples of the form $c(o_1, o_2, m)$, where o_1 is the object that is executing, o_2 is the object responsible for executing the call, and m is the name of the invoked method. These cost centers are symbolic expressions that will be part of the upper bound computed by the analyzer. Let us see an example.

Example 7 (number of tasks). For the code in Fig. 1, cost analysis infers that the number of iterations of the loop in extend (at L12) is bounded by the expression $\mathsf{nat}(s)$, where $\mathsf{nat}(e)$ returns e if $e > 0$ and 0 otherwise. Since the size of l is increased within the loop at L12, the maximum number of iterations for the loop at L25 is produced in the last call to process. Recall that l represents the term size of the list l (see Def. 1), and it counts 2 units for each element in the list. Therefore, each iteration of the loop at L25 increments the term size of the list in 2 units and, consequently, the last call to process is done with a list of size $l+2*s$. The loop in process (L25) traverses the list received as argument consuming 2 size units per iteration. Therefore, the expression $(l + 2 * s)/2 = l/2 + s$ bounds the number of iterations of such loop. As process is called $\mathsf{nat}(s)$ times, $\mathsf{nat}(s) * \mathsf{nat}(l/2 + s)$ bounds the number of times that the body of the loop at L25 is executed. Then, by applying the number of tasks cost model we obtain the following expression that bounds the number of tasks spawned:

$$\begin{aligned}\mathcal{U}^t_{\mathsf{extend}}(l, s) = {} & c(o_x, o_y, \mathsf{add}) * \mathsf{nat}(s) + \\ & c(o_x, o_z, \mathsf{process}) * \mathsf{nat}(s) + \\ & c(o_z, o_y, \mathsf{foo}) * (\mathsf{nat}(s) * \mathsf{nat}(l/2{+}s))\end{aligned}$$

From the upper bounds on the tasks spawned, we can obtain a range of useful information: (1) If we are interested in the number of communications for the whole program, we just replace all expressions $c(o_1, o_2, m)$ by 1. (2) Replacing all cost centers of the form $c(o, \_, \_)/c(\_, o, \_)$ by 1 for the object o and the remaining ones by 0, we obtain an upper-bound on the number of tasks spawned from/in o. We use, respectively, $\mathcal{U}_m|_{o\rightarrow}$ and $\mathcal{U}_m|_{\rightarrow o}$ to refer to the UB on the outgoing/incoming tasks. (3) Replacing $c(o_1, o_2, \_)$ by 1 for selected objects and the remaining ones by 0, we can see the tasks spawned by o_1 in o_2, denoted by $\mathcal{U}_m|_{o1\rightarrow o_2}$. (4) If we are interested in a particular method p, we can replace $c(\_, \_, p)$ by 1 and the rest by 0, we use $\mathcal{U}_m|_{\xrightarrow{p}}$ to denote it.

Example 8 (number of tasks restriction). Given the upper bound of Ex. 7, the number of tasks spawned from o_x to o_y is captured by replacing $c(o_x, o_y, \_)$ (the method is not relevant) by 1 and the rest by 0. Then, we obtain the expression $\mathcal{U}^t_{\mathsf{extend}}|_{o_x\rightarrow o_y} = \mathsf{nat}(s)$, which shows that we have one task for each iteration of the loop at L13. We can also obtain an upper bound on the number of tasks from o_z to o_y, $\mathcal{U}^t_{\mathsf{extend}}|_{o_z\rightarrow o_y} = \mathsf{nat}(s) * \mathsf{nat}(l/2{+}s)$. The number of tasks spawned using method foo are captured by $\mathcal{U}^t_{\mathsf{extend}}|_{\xrightarrow{\mathsf{process}}} = \mathsf{nat}(s)$.

4.3 Inference of amount of transmitted data

Our goal now is to infer, not only the number of tasks spawned, but also the sizes of the arguments in the task invocation and of the returned values. Formally, this

is done by extending the previous cost model to include data sizes as well. We rely on two auxiliary functions. Given a variable x at a certain program point, function $\alpha(x)$ returns the term size of this variable at this point, as defined in Sec. 3. Besides, after spawning a task, we are interested in knowing whether the result of executing the task is retrieved, and in such case we accumulate the size of the return value. This information is computed by a *may-happen-in-parallel* analysis [5] which allows us to know to which task a future variable is associated. Thus, we can assume the existence of a function $hasGet(pp)$ which returns if the result of the task spawned at program point pp is retrieved by a get instruction. Now, we define a new cost model that counts the sizes of the data transferred in each communication by relying on the two functions above.

Definition 6 (data sizes cost model). *Given a program point pp we define the cost model $\mathcal{M}^d(ins)$ as a function which returns $sc(ins)$ if $pp : ins \equiv r = y!m(\overline{x}) \wedge o_1 \neq o_2 \wedge o_1 \in pt(pp, this) \wedge o_2 \in pt(pp, y)$, and 0, otherwise; where*

$$sc(ins) = \begin{cases} c(o_1,o_2,m)*(\mathcal{I} + \sum_{x_i \in \overline{x}} \mathsf{nat}(\alpha(x_i))) + c(o_2,o_1,m)*(\mathcal{I} + \mathsf{nat}(\alpha(r))) & \textit{if } hasGet(pp) \\ c(o_1,o_2,m)*(\mathcal{I} + \sum_{x_i \in \overline{x}} \mathsf{nat}(\alpha(x_i))) & \textit{otherwise} \end{cases}$$

Observe that the above cost model extends the one in Def. 5 as it extends the number of tasks cost model with the sizes of the data transmitted. Intuitively, as any call always transfers its input arguments, their size is always included (second case). However, the size of the returned information is only included when there exists a get instruction that retrieves this information (first case). In each case, we include the size for sending the messages $\mathcal{I}$. Note that the cost centers reflect the direction of the transmission, $c(o_1, o_2, m)$ corresponds to a transmission from o_1 to o_2 through a call to m, whereas $c(o_2, o_1, m)$ corresponds to the information returned by o_2 in response to a call to m spawned by o_1. If needed, call and return cost centers can be distinguished by marking the method name, e.g., m for calls and m^r for returns. As already mentioned, nat denotes the positive value of an expression. We wrap the size of each argument using nat because this way the analyzer treats them as an expression whose cost we want to maximize (the technical details of the maximization operation can be found in [4]). Therefore, the upper bound inferred by the analyzer using this cost model already provides the overall information (i.e., number of tasks spawned and maximum size of the data transmitted).

Example 9 (data sizes cost model). Let us see the application of the cost model to the calls at L16, L13 and L26. At L16 we have the instruction z!process(l). As the program does not retrieve any information from process(l), the function $hasGet(L16)$ returns false, and thus we only include the calling data. Then, using the points-to information in Ex. 6, the application of $\mathcal{M}^d$ at L16 returns: $\mathcal{M}^d(\mathsf{z!process(l)}) = c(o_x, o_z, \mathsf{process}) * \mathcal{I} + \mathsf{nat}(\alpha(l))$. As l is a data structure and it is modified within the loop, $\alpha(l)$, returns the term size of l. Observe that the expression captures, not only the objects and the method involved in the

call within the cost center, but also the amount of data transferred in the call, $\mathsf{nat}(\alpha(l))$. The application of $\mathcal{M}^d$ to the call at L13, $\mathsf{f} = \mathsf{y!add(l,5)}$, returns the expression:

$$\begin{aligned}\mathcal{M}^d(\mathsf{f{=}y!add(l_0{,}5)}) = {} & c(o_x, o_y, \mathsf{add}) * (\mathcal{I} + \mathsf{nat}(\alpha(l_0)) + \mathsf{nat}(\alpha(5)))+ \\ & c(o_y, o_x, \mathsf{add}) * (\mathcal{I} + \mathsf{nat}(\alpha(f)))\end{aligned}$$

In this case, at L15 we have a get for the call at L13, so $hasGet(L13) = true$. Note that we use l_0 to refer to the value of l at the beginning of the loop and l to refer to the value of the list after calling add. The application of $\alpha(5)$ returns 1, as it is a basic type (counting as one constructor). The call at L27 returns the expression $c(o_y, o_z, \mathsf{foo}) * (\mathcal{I} + \mathsf{nat}(\alpha(h)))$.

As we have explained above, the size of a data structure might depend on the input arguments that in turn can be modified along the program execution. Consequently, if we are in a loop, for the same program point, the amount of data transferred in one call can be different for each iteration of the loop. Soundness of the cost analysis ensures that it provides the worst possible size in such case. Technically, it is done by maximizing [4] the expressions inside nat within their calling context.

Example 10 (data sizes upper bound). Once the cost model is applied to all instructions in the program, we obtain a set of recursive equations which define the transmission data sizes within the locations in the program. After solving such equations using [4], we obtain the following expression which defines the transmission data sizes of any execution starting from extend, denoted by $\mathcal{U}^d_{\mathsf{extend}}$:

$$\begin{aligned}\mathcal{U}^d_{\mathsf{extend}}(l, s) = {} & c(o_x, o_y, \mathsf{add}) * \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2 - 2) + 1)+ & \textcircled{1} \\ & c(o_y, o_x, \mathsf{add}) * \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2))+ & \textcircled{2} \\ & c(o_x, o_z, \mathsf{process}) * \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2))+ & \textcircled{3} \\ & c(o_z, o_y, \mathsf{foo}) * (\mathsf{nat}(s) * \mathsf{nat}(l/2{+}s)) * (\mathcal{I} + 1) & \textcircled{4}\end{aligned}$$

The expression at ① includes the transmission from o_x to o_y. The worst case size of the list at this point is $\mathsf{nat}(l+s*2-2)$, this is because initially the list has size $\mathsf{nat}(l)$ and at each iteration of the loop, the size is increased in method add by two elements: Cons and an integer value. As the loop performs s iterations, in the last invocation to add it has length $l + (s-1) * 2$. This size is assumed for all loop iterations (worst case size), hence we infer that the maximum data size transmitted from o_x to o_y is $\mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2 - 2) + 1)$, the 1 is due to the second argument of the call (an integer). At ②, o_x receives from o_y the same list, but including the last element, that is $\mathsf{nat}(l + s * 2)$. The same list is obtained at ③. In ④, the cost is constant in all iterations (1 integer).

As already mentioned in Sec. 4.2, the fact that cost centers are symbolic expressions allows us to extract different pieces of information regarding the amount of data transferred between the different abstract locations involved in the communications. With $\mathcal{U}^d$ we can infer, not only an upper-bound on the total amount of data transferred along the program execution, but also the size of the data transferred between two objects, or the incoming/outgoing data sent/received by a particular object.

				Nodes			Methods			Pairs		
Benchmark	loc	$\#_c$	T	$\%^n_M$	$\%^n_m$	$\%^n_a$	$\%^m_M$	$\%^m_m$	$\%^m_a$	$\%^p_M$	$\%^p_m$	$\%^p_a$
BBuffer	200	17	829	25.7	0.6	16.3	43.9	0.1	6.2	7.3	0.0	0.7
MailServer	119	13	693	30.0	4.4	15.4	27.3	0.5	10.0	8.7	0.0	0.6
Chat	302	10	171	40.5	7.5	20.0	12.7	0.1	3.0	9.6	0.0	1.1
DistHT	146	9	1204	48.0	3.0	18.7	40.7	0.3	10.0	8.0	0.0	0.9
BookShop	366	10	3327	58.7	3.9	23.9	23.6	0.1	8.3	29.5	0.0	1.5
PeerToPeer	263	19	62575	27.7	0.1	15.6	20.6	0.1	5.8	5.9	0.0	0.5

Table 1. Experimental results (times in ms)

Example 11 (data sizes restriction). From $\mathcal{U}^d_{\mathsf{extend}}(l,s)$, using the cost centers as we have explained in Ex. 8, we can extract different types of information about the data transferred. For instance, we can bound the size of the outgoing data from location x:

$$\mathcal{U}^d_{\mathsf{extend}}(l,s)|_{o_x\rightarrow} = \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2 - 2) + 1) + \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2))$$

Or the incoming data sizes for the location y:

$$\mathcal{U}^d_{\mathsf{extend}}(l,s)|_{\rightarrow o_y} = \mathsf{nat}(s) * (\mathcal{I} + \mathsf{nat}(l + s * 2 - 2) + 1) + (\mathsf{nat}(s) * \mathsf{nat}(l/2{+}s)) * (\mathcal{I} + 1)$$

Theorem 1 (soundness). *Let P be a program and l_1, l_2 location identifiers. Let $\mathcal{O}$ be the object names computed by a points-to analysis of P. Let o_1, o_2 be the abstractions of l_1, l_2 in $\mathcal{O}$. Then, given a trace $\mathcal{T}$ from P with arguments $\overline{x}$ we have that*

$$trans(\mathcal{T}|_{l_1 \xrightarrow{m} l_2}) \leq \mathcal{U}^d_P(\overline{x})|_{o_1 \xrightarrow{m} o_2}.$$

5 Experimental Results

We have implemented our analysis in SACO [2] and applied it to some typical examples of distributed systems: BBuffer, a bounded-buffer for communicating several producers and consumers; MailServer, a client-server distributed system; Chat, a chat application; DistHT, a distributed hash table; BookShop, a web shop client-server application; and PeerToPeer, a peer-to-peer network with a set of interconnected peers. Experiments have been performed on an Intel Core i7 at 3.4GHz with 8GB of RAM, running Ubuntu 12.04.

We have applied our analysis and evaluated the upper bound expressions for different combinations of concrete input values so as to obtain some quantitative information about the analysis. Table 1 summarizes the results obtained. Columns Benchmark and loc show, resp. the name and the number of program lines of the benchmark. Column $\#_c$ displays the number of locations identified by the analysis. Column T shows the time to perform the inference of the transmission data sizes. We have studied the transmission data sizes among each pair of locations identified by the points-to analysis. We have studied data transmission from three points of view: (1) from a location with the rest of the program, (2) from a method, and (3) among pairs of locations. In case (1), we try to

identify potential bottlenecks in the communication, i.e., those locations that produce/consume most of the data in the benchmark. Also, we want to observe locations that do not have much communication. In the former, such locations should have a fast communication channel, while in the latter we can still have a good response time with slower bandwidth conditions. Columns $\%^n_M$, $\%^n_m$, $\%^n_a$ show, respectively, the percentage of the location that accumulates more traffic (incoming + outgoing) w.r.t. the total traffic in the system, for the location with less traffic, and the average for the traffic of all locations. Similarly, columns $\%^p_M$, $\%^p_m$, $\%^p_a$ show, for case (3), which is the percentage of the total traffic transmitted by the pair of locations that have more traffic, by the pair with less traffic and the average between the traffic of all pairs, respectively. Finally, regarding case (2), columns under Methods show similar information but taking into account the task that performs the communication, i.e., the percentage of the traffic transmitted by the task that transmits more (resp., less) amount of data, $\%^m_M$ (resp., $\%^m_m$), and the average of the transmissions performed by each task ($\%^m_a$).

We can observe in the table that our analysis is performed in a reasonable time. One important issue is that we only have to perform the analysis once, and the information can be extracted later by evaluating the upper bound with different parameters and focusing in the communications of interest. In the columns for the locations, we can see that all benchmarks are relatively well distributed. The average of the data transmitted per location is under 25% for all benchmarks. BookShop is the benchmark which could have a communication bottleneck as it accumulates in a single location 58.7% of the total traffic. Regarding methods, it is interesting to see that for all benchmarks no method accumulates more than 45% of the total traffic. Moreover, the table shows that in all benchmarks there is at least one method that requires less than 0.5%, in most cases this method (or methods) is an object constructor. Regarding pairs of locations, in all benchmarks there is at least one pair of locations that do not communicate, $\%^p_m = 0$ for all benchmarks. This is an expected result, as it is quite often to have pairs of locations which do not communicate in a distributed program. Our experiments thus confirm that transmission among pairs of locations is relatively well distributed, as in most benchmarks, except for BookShop, the pair with highest traffic requires less than 10% of the total traffic.

6 Conclusions

We have presented a static analysis to soundly approximate the amount of data transmitted among the locations of a distributed system. This is an important contribution to be able to infer the response times of distributed components. In particular, if one knows the bandwidth conditions among each pair of locations, we can infer the time required to transmit the data and to retrieve the result. This time should be added to the time required to carry out the computation at each location, which is an orthogonal issue. Conversely, we can use our analysis to establish the bandwidth conditions required to ensure a certain response time. Technically, our analysis is formalized by defining a new cost model which

captures only the data transmission aspect of the application. This cost model can be plugged into a generic cost analyzer for distributed systems, that directly returns an upper bound on the transmission data sizes, without requiring any modification to the other components of the cost analyzer.

Acknowledgments. 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

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
3. 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.
4. 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.
5. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
6. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
7. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM Press, 1978.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
10. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
11. 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.
12. Jonathan K. Lee and Jens Palsberg. Featherweight x10: a core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, 2010. 1693459.
13. 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.
14. M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proc. of POPL'97*, pages 1–14, Paris, France, January 1997. ACM.
15. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
16. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

Appendix F

Article *Non-Cumulative Resource Analysis*, [11]

Non-Cumulative Resource Analysis Author's version**

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

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

Abstract. Existing cost analysis frameworks have been defined for cumulative resources which keep on increasing along the computation. Traditional cumulative resources are execution time, number of executed steps, amount of memory allocated, and energy consumption. Non-cumulative resources are acquired and (possibly) released along the execution. Examples of non-cumulative cost are memory usage in the presence of garbage collection, number of connections established that are later closed, or resources requested to a virtual host which are released after using them. We present, to the best of our knowledge, the first generic static analysis framework to infer an *upper bound* on the *peak cost* for non-cumulative types of resources. Our analysis comprises several components: (1) a pre-analysis to infer when resources are being used simultaneously, (2) a *program-point* resource analysis which infers an upper bound on the cost at the points of interest (namely the points where resources are acquired) and (3) the elimination from the upper bounds obtained in (2) of those resources accumulated that are not used simultaneously. We report on a prototype implementation of our analysis that can be used on a simple imperative language.

1 Introduction

Cost analysis (a.k.a. resource analysis) aims at statically (without executing the program) inferring *upper bounds* on the resource consumption of the program as functions of the input data sizes. Traditional resources (e.g., time, steps, memory allocation, number of calls) are *cumulative*, i.e., they always increase along the execution. Ideally, a cost analysis framework is *generic* on the type of resource that the user wants to measure so that the resource of interest is a parameter of the analysis. Several generic cost analysis frameworks have been defined for cumulative resources using different formalisms. In particular, the classical framework based on recurrence relations has been used to define a cost

[2] Appeared at *Proc. 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), London, UK, April 11-18, 2015.* The final publication is available at http://link.springer.com/chapter/10.1007/978-3-662-46681-0_6

analysis for a Java-like language [2]; approaches based on program invariants are defined in [10,13]; type systems have been presented in [14].

Non-cumulative resources are first acquired and then released. Typical examples are memory usage in the presence of garbage collection, maximum number of connections established simultaneously, the size of the stack of activation records, etc. The problem is nowadays also very relevant in *virtualized* systems, as in cloud computing, in which resources are acquired when needed and released after being used. It is recognized that non-cumulative resources introduce new challenges in resource analysis [5,11]. This is because the resource consumption can increase and decrease along the computation, and it is not enough to reason on the final state of the execution, but rather the upper bound on the cost can happen at any intermediate step. We use the term *peak cost* to denote such maximum cost of the program execution for non-cumulative resources.

While the problem of inferring the peak cost has been studied in the context of memory usage for specific models of garbage collection [5,8,11], a generic framework to estimate the non-cumulative cost does not exist yet. The contribution of this paper is a generic resource analysis framework for a today's imperative language enriched with instructions to acquire and release resources. Thus, our framework can be instantiated to measure any type of non-cumulative resource that is acquired and (optionally) freed. The analysis is defined in two steps which are our main contributions: (1) We first infer the sets of resources which can be in use simultaneously (i.e., they have been both acquired and none of them released at some point of the execution). This process is formalized as a static analysis that (over-)approximates the sets of acquire instructions that can be in use simultaneously, allowing us to capture the simultaneous use of resources in the execution. (2) We then perform a *program-point* resource analysis which infers an upper bound on the cost at the points of interest, namely the points at which the resources are acquired. From such upper bounds, we can obtain the peak cost by just eliminating the cost due to acquire instructions that do not happen simultaneously with the others (according to the analysis information gathered at step 1). Additionally, we describe an extension of the framework which can improve the accuracy of the upper bounds by accounting only once the cost introduced at program points where resources are allocated and released repeatedly. Finally, we illustrate how the framework can be extended to get upper bounds for programs that allocate different kinds of resources.

We demonstrate the accuracy and feasibility of our approach by implementing a prototype analyzer for a simple imperative language. Preliminary experiments show that the non-cumulative resource analysis achieves gains up to 92.9% (on average 53.9%) in comparison to a cumulative resource analysis. The analysis can be used online from a web interface at `http://costa.ls.fi.upm.es/noncu`.

2 The Notion of Peak Cost

We start by defining the notion of peak cost that we aim at over-approximating by means of static analysis in the concrete setting.

$$(1)\ \frac{r = eval(\mathsf{e}, tv), tr' = tr[y \mapsto \langle r, a_{pp}\rangle], H' = H \cup \{\!|\langle id, y, a_{pp}, r\rangle|\!\}}{\langle id, m, pp \equiv \mathsf{y} = \mathsf{acquire\ (e)}; s, tv, tr\rangle \cdot A; H \ \rightsquigarrow\ \langle id, m, s, tv, tr'\rangle \cdot A; H'}$$

$$(2)\ \frac{\langle r, a_{pp'}\rangle = tr(y), tr' = tr[y \mapsto \bot], H' = H \setminus \{\!|\langle id, y, a_{pp'}, r\rangle|\!\}}{\langle id, m, pp \equiv \mathsf{release\ y}; s, tv, tr\rangle \cdot A; H \ \rightsquigarrow\ \langle id, m, s, tv, tr'\rangle \cdot A; H'}$$

Fig. 1. Language Semantics for resource allocation and release

2.1 The Language

The framework is developed on a language which is deliberately simple to define the analysis in a clear way. Complex features of modern languages like mutable variables, class, inheritance, exceptions, etc. must be considered by the underlying resource analysis used as a black box by our approach (and there are a number of approaches to handle them [2,5,10]). Thus they are handled implicitly in our setting. For the sake of simplicity, the set *Types* is defined as $\{\mathsf{int}\}$. We have *resource* variables used to refer to the resources allocated by an acquire instruction. A program consists of a set of methods whose definition takes the form $t\ m\ (t_1 v_1, \ldots t_n v_n)\{s\}$ where $t \in$ *Types* is the type returned by the method, $v_1, \ldots, v_n$ are the input parameters of types $t_1, \ldots, t_n \in$ *Types* and s is a sequence of instructions that adheres to the following grammar:

$$e ::= x \mid n \mid e + e \mid e * e \mid e - e \qquad b ::= e > e \mid e == e \mid b \wedge b \mid b \vee b \mid !b \qquad s ::= i \mid i; s$$
$$i ::= x{=}e \mid x{=}m(\overline{z}) \mid \mathsf{return}\ x \mid \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{while}\ b\ \{s\} \mid y = \mathsf{acquire}\ (e) \mid \mathsf{release}\ y$$

We assume that resource variables, named y, are local to methods and they cannot be passed as input parameters nor returned by methods (otherwise tracking such references is more complex, while it is not relevant to the main ideas in the paper). We assume that the program includes a main($\overline{\mathsf{x}}$) method, where $\overline{\mathsf{x}}$ are the input parameters, from which the execution starts. The instruction y = acquire (e) allocates the amount of resources stated by the expression e. The instruction release y releases the resources allocated at the last acquire associated to y. If a resource variable is reused without releasing its resources, the reference to such resources is lost and they cannot be released any longer.

Example 1. Fig. 2 shows to the left a method m (abbreviation of main) that allocates resources at lines 2 (L2 for short) and L4. The resources allocated at L2 are released at L5. In addition, method m invokes method q at L3 and L6. For simplicity, we assume that m is called using positive values for n and s and the expressions k_1, k_2, k_3 are constant integer values. As it is not relevant, we do not include the return instruction at the end of the methods. Method q executes a while loop where k_2 units are allocated at L10 and such resources are not released. Thus, these resources *escape* from the scope of the loop and the method, i.e., they *leak* upon exit of the loop and return of the method. Besides, the program allocates w units at L11. As we have two calls to q, the input parameter w will take the value s or s+4. The resources allocated at L11 are released at L12 and do not escape from the loop execution. In addition, at L15 we have an additional, non-released, acquire of k_3 units.

A *program state* is of the form $AS;H$, where AS is a stack of *activation records* and H is a *resource handler*. Each activation record is of the form $\langle id, m, s, tv, tr\rangle$,

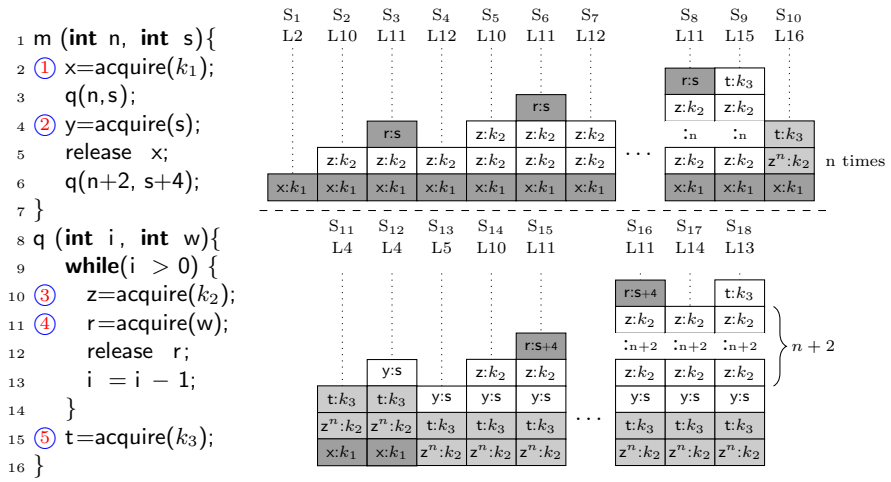


Fig. 2. Running Example

where id is a unique identifier, m is the name of the method, s is the sequence of instructions to be executed, tv is a variable mapping and tr is a resource variable mapping. When resources are allocated in m, tr maps the corresponding resource variable to a tuple of the form $\langle r, a_{pp}\rangle$, where r is the amount of resources allocated and a_{pp} is the program point of the instruction where the resources have been allocated. The resource handler H is a multiset which stores the resources allocated so far, containing elements of the form $\langle id, y, a_{pp}, r\rangle$, where id is the activation record identifier, y is the variable name, a_{pp} is the program point of the acquire and r is the amount of resources allocated. Fig. 1 shows, in a rewriting-based style, the rules that are relevant for the resource consumption. The semantics of the remaining instructions is standard. Intuitively, rule (1) evaluates the expression e and adds a new element to H. As H stores the resources allocated so far, it might contain identical tuples. Moreover, the resource variable mapping tr is updated with variable y linked to $\langle r, a_{pp}\rangle$. Rule (2) takes the information stored in tr for y, i.e. $\langle r, a_{pp}\rangle$, and removes from H one instance of the corresponding element. In addition, variable y is updated to point to $\perp$, which means that y does not have any resources associated. When the execution performs a release on a variable that maps to $\perp$ (because no acquire has been performed or because it has already been released), the resources state is not modified. Execution starts from a main method and an initial state $S_0 = \langle 0, \mathsf{main}, body(\mathsf{main}), tv(\overline{x}), \emptyset\rangle; \emptyset$, where $tv(\overline{x})$ is the variable mapping initialized with the values of the input parameters. Complete executions are of the form $S_0 \rightsquigarrow S_1 \rightsquigarrow \ldots \rightsquigarrow S_n$ where S_n corresponds to the last state. Infinite traces correspond to non-terminating executions.

Example 2. To the right of Fig. 2 we depict the evolution of the resources accumulated in H. We use S_i, to refer to the execution state i and, below each state, we include the program line which is executed at such state. For each state we show the elements stored in H but, for simplicity, we do not include in the figure

the *id* nor a_{pp}. At S_1, H accumulates k_1 units due to the acquire at L2. S_2, S_3 and S_4 depict H along the first iteration of the loop, where k_2 units are acquired and not released from z. Moreover, within the loop, s units are acquired at L11 and released from r at L12. At S_5, which corresponds to the second iteration of the loop, we reuse the resource variable z and we have two identical elements in H. As the loop iterates n times, at the last iteration (S_9) we have $(n{-}1){*}k_1$ units that have lost their reference. Additionally, k_3 extra units pointed by t are allocated at S_9. At S_{10}, which corresponds to the end of the execution of the method, $n{*}k_2{+}k_3$ units escape from the first execution of q and they are no longer available to be released. We represent such escaped resources with light grey color. For brevity, we use $\mathsf{z}^{n:k_2}$ to represent n instances of the element $\mathsf{z}{:}k_2$. At S_{12} we acquire s resources and we release the k_1 units pointed by x at S_{13}. At S_{14} we start a new execution of method q.

2.2 Definition of Peak Cost

Let us formally define the notion of *peak cost* in the concrete setting. The peak cost corresponds to the maximum amount of resources that are used simultaneously. We use H_i to refer to the multiset H at S_i, and we use R_i to denote the amount of resources contained in H_i, i.e., $R_i = \sum\{r \mid \langle \_, \_, \_, r\rangle \in H_i\}$. By '_', we mean any possible value. In the next definition, we use R_i to define the notion of *peak cost* for an execution trace.

Definition 1 (concrete peak cost). *The* peak cost *of an execution trace* $t{\equiv}S_0{\rightsquigarrow}S_n$ *of a program* P *on input values* $\overline{x}$ *is defined as* $\mathcal{P}(\overline{x}){=}max(\{R_i \mid S_i{\in}t\})$.

Example 3. According to the evolution of H shown to the right of Fig. 2, the maximum value of R_i could be reached at four different states, S_8, S_{12}, S_{16} and S_{18}. We ignore those states where H is subsumed by other states as they cannot be maximal. For instance, states S_1 to S_7 or S_9 are subsumed by S_8; or S_{12} contains S_{10}, S_{11} and S_{13}. Thus, $\mathcal{P}(n,s){=}max(R_8, R_{12}, R_{16}, R_{18})$, where $R_8 = k_1{+}n{*}k_2{+}s$, $R_{12} = k_1{+}n{*}k_2{+}k_3{+}s$, $R_{16} = n{*}k_2{+}k_3{+}s{+}(n{+}2){*}k_2{+}(s{+}4)$, and $R_{18}{=}n{*}k_2{+}k_3{+}s{+}(n{+}2){*}k_2{+}k_3$. Thus, the peak cost of the example depends not only on the input parameters n, s, but also on the values of k_1, k_2, k_3.

3 Simultaneous Resource Analysis

The *simultaneous resource analysis* (SRA) is used to infer the sets of acquire instructions that can be simultaneously in use. The abstract state of the SRA consists of two sets $\mathcal{C}$ and $\mathcal{H}$. The set $\mathcal{C}$ contains elements of the form $y{:}a_{pp}$ indicating that the resource variable y is linked to the acquire instruction at program point pp. Since it is not always possible to relate the acquire instruction to its corresponding resource variable, we use $\star{:}a_{pp}$ to represent that some resources have been acquired at a_{pp} but the analysis has lost the variable linked to a_{pp}. The set $\mathcal{H}$ is a set of sets, such that each set contains those a_{pp} that are simultaneously alive in an abstract state of the analysis. Let us introduce some notation. We use $\ddot{m}$ to refer to the program point after the return instruction

$$
\begin{array}{ll}
(1) & \tau(pp : \mathsf{y{=}acquire(\_)}, \langle \mathcal{C}, \mathcal{H}\rangle) = \langle \mathcal{C}[y{:}a_{pp'} / \star{:}a_{pp'}] \cup \{y{:}a_{pp}\}, \mathcal{H} \uplus \{\mathcal{A}(\mathcal{C}) \cup \{a_{pp}\}\}\rangle \\
(2) & \tau(pp : \mathsf{release\ y}, \langle \mathcal{C}, \mathcal{H}\rangle) = \langle \mathcal{C} \setminus \{y{:}a_{pp}\}, \mathcal{H}\rangle \\
(3) & \tau(pp : \mathsf{m(\_)}, \langle \mathcal{C}, \mathcal{H}\rangle) = \langle \mathcal{C} \cup \mathcal{C}_{\ddot{m}}[x{:}a_{pp'} / \star{:}a_{pp'}], \mathcal{H} \uplus \{\mathcal{A}(\mathcal{C}) \cup M \mid M \in \mathcal{H}_{\ddot{m}}\}\rangle \\
(4) & \tau(pp : b, \langle \mathcal{C}, \mathcal{H}\rangle) = \langle \mathcal{C}, \mathcal{H}\rangle
\end{array}
$$

Fig. 3. Transfer Function of the Simultaneous Resource Analysis

of method m. We use $\mathcal{C}_{pp}$ (resp. $\mathcal{H}_{pp}$) to denote the value of $\mathcal{C}$ (resp. $\mathcal{H}$) after processing the instruction at program point pp. $\mathcal{A}(\mathcal{C})$ is the set $\{a_{pp} \mid \_{:}a_{pp} \in \mathcal{C}\}$ that contains all a_{pp} in $\mathcal{C}$. The operation $\mathcal{H}_1 \uplus \mathcal{H}_2$, where $\mathcal{H}_1$ and $\mathcal{H}_2$ are sets of sets, first applies $\mathcal{H} = \mathcal{H}_1 \cup \mathcal{H}_2$, and then removes those sets in $\mathcal{H}$ that are contained in another set in $\mathcal{H}$.

The analysis of each method m abstractly executes its instructions, by applying the transfer function τ in Fig. 3, such that the abstract state at each program point describes the status of all acquire instructions executed so far. The set $\mathcal{C}$ is used to infer the local effect of the acquire and release instructions within a method. The set $\mathcal{H}$ is used to accumulate the information of the acquire instructions that might have been in use simultaneously. Let us explain the different cases of the transfer function τ. The execution of acquire, case (1), links the acquire to the resource variable y by adding $\{y{:}a_{pp}\}$ to $\mathcal{C}$. As a resource variable can only point to one acquire instruction, in (1) we update any existing $y{:}a_{pp'}$ by removing the previous link to y and replacing it by $\star$. In addition, rule (1) performs the operation $\{\mathcal{A}(\mathcal{C}) \cup \{a_{pp}\}\} \uplus \mathcal{H}$ to capture in $\mathcal{H}$ the acquired resources simultaneously in use at this point. In (2) we remove the last acquire instruction pointed to by the resource variable y. When a method is invoked (rule (3)), we add to $\mathcal{C}$ those resources that might escape from m ($\mathcal{C}_{\ddot{m}}$) but replacing their resource variables in m by $\star$ (as resource variables are local). Additionally, at (3), all sets in $\mathcal{H}_{\ddot{m}}$ are joined with $\mathcal{A}(\mathcal{C})$ to capture the resources that might have been simultaneously alive in the execution of m. The resulting sets of such operation are added to $\mathcal{H}$. We define the $\sqcup$ operation between two abstract states $\langle \mathcal{C}_1, \mathcal{H}_1\rangle \sqcup \langle \mathcal{C}_2, \mathcal{H}_2\rangle$ as $\langle \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{H}_1 \uplus \mathcal{H}_2\rangle$. The analysis of while loops requires iterating until a fixpoint is reached. As the number of acquire instructions and the number of resource variables in the program are finite, widening is not needed.

Example 4. Let us apply the SRA to the running example. To avoid cluttering the expressions, instead of the line numbers, we use a_i to refer to the acquire at the program point marked with ⓘ in Fig. 2. For instance, a_1 refers to the acquire marked with ① at L2. We use $\mathcal{C}_l$ (resp. $\mathcal{H}_l$) to denote the set $\mathcal{C}$ (resp. $\mathcal{H}$) at line l. Let us see the results of the SRA for some selected program points.

$$
\begin{array}{ll}
\mathcal{C}_2 = \{x{:}a_1\} & \mathcal{H}_2 = \{\{a_1\}\} \\
\mathcal{C}_3 = \{x{:}a_1, \star{:}a_3, \star{:}a_5\} & \mathcal{H}_3 = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5\}\} \\
\mathcal{C}_4 = \{x{:}a_1, \star{:}a_3, \star{:}a_5, y{:}a_2\} & \mathcal{H}_4 = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}\} \\
\mathcal{C}_5 = \{\star{:}a_3, \star{:}a_5, y{:}a_2\} & \mathcal{H}_5 = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}\} \\
\mathcal{C}_6 = \mathcal{C}_{\ddot{m}} = \{\star{:}a_3, \star{:}a_5, y{:}a_2\} & \mathcal{H}_6{=}\mathcal{H}_{\ddot{m}}{=}\{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}, \{a_2, a_3, a_4, a_5\}\,(*)\} \\
\mathcal{C}_{10} = \{\star{:}a_3, z{:}a_3\} & \mathcal{H}_{10} = \{\{a_3, a_4\}\} \\
\mathcal{C}_{11} = \{\star{:}a_3, z{:}a_3, r{:}a_4\} & \mathcal{H}_{11} = \{\{a_3, a_4\}\} \\
\mathcal{C}_{12} = \mathcal{C}_{14} = \{\star{:}a_3, z{:}a_3\} & \mathcal{H}_{12} = \mathcal{H}_{14} = \{\{a_3, a_4\}\} \\
\mathcal{C}_{15} = \mathcal{C}_{\ddot{q}} = \{\star{:}a_3, z{:}a_3, t{:}a_5\} & \mathcal{H}_{15} = \mathcal{H}_{\ddot{q}} = \{\{a_3, a_4\}, \{a_3, a_5\}\}
\end{array}
$$

We can see that $\mathcal{C}_{11}$ is the only program point where a_4 is alive as it is released at L12. On the contrary, as a_3 is not released within the loop, we include $\star{:}a_3$ in $\mathcal{C}_{10}$–$\mathcal{C}_{14}$, and it escapes from the loop and from q. As $\mathcal{H}$ gathers all a_{pp} that might be alive at any program point, when the fixpoint is reached, $\mathcal{H}_{10} - \mathcal{H}_{14}$ contain the set $\{a_3, a_4\}$. The computation of $\mathcal{H}_{\ddot{q}}$ is done by means of the operation $\mathcal{A}(\mathcal{C}_{\ddot{\mathsf{q}}}) \uplus \mathcal{H}_{14}$, that is, $\mathcal{H}_{\ddot{q}}{=}\{\{a_3, a_5\}\} \uplus \{\{a_3, a_4\}\}{=}\{\{a_3, a_5\}, \{a_3, a_4\}\}$, capturing that a_3, a_4, a_5 are not simultaneously in use at any state of q. Moreover, we can see in $\mathcal{C}_{\ddot{\mathsf{q}}}$ that the resources allocated at a_3 and a_5 escape from the execution of q. Let us continue with the computation of $\mathcal{C}_3$ and $\mathcal{H}_3$. Firstly, $\star{:}a_3$ and $\star{:}a_5$ are added to $\mathcal{C}_3$. Secondly, $\mathcal{H}_3$ is computed by adding $\mathcal{C}_2{=}\{a_1\}$ to all sets in $\mathcal{H}_{\ddot{\mathsf{q}}}$. To compute $\mathcal{C}_4$, the analysis adds $y{:}a_2$ to $\mathcal{C}_3$. The computation of $\mathcal{H}_4$ adds $\{a_1, a_3, a_5, a_2\}$ to $\mathcal{H}_3$, and replaces $\{a_1, a_3, a_5\}$ because it is a subset of $\{a_1, a_3, a_5, a_2\}$. Finally, to obtain $\mathcal{H}_6$, the set $\mathcal{A}(\mathcal{C}_6){=}\{a_3, a_5, a_2\}$ is added to the sets in $\mathcal{H}_{\ddot{\mathsf{q}}}$, resulting in the set $T = \{\{a_2, a_3, a_4, a_5\}, \{a_2, a_3, a_5\}\}$. Then $\mathcal{H}_6$ is obtained by computing $\mathcal{H}_5 \uplus T$. Note that $\{a_2, a_3, a_5\}$ is not in $\mathcal{H}_6$ as it is contained in a set of $\mathcal{H}_5$.

Theorem 1 (soundness). *Given an execution trace $t \equiv S_0 \rightsquigarrow \ldots \rightsquigarrow S_n$ of a program P on input values $\overline{x}$, for any state $S_i \in t$, we have that:*
(a) $\exists\, \mathbb{H} \in \mathcal{H}_{\ddot{\mathsf{main}}}.\ A(H_i) \subseteq \mathbb{H}$ *where* $A(H_i) = \{a_{pp} \mid \langle \_, \_, a_{pp}, \_\rangle \in H_i\}$;
(b) if $\exists \langle \_, \_, a_{pp}, \_\rangle \in H_n$ *then* $\_{:}a_{pp} \in \mathcal{C}_{\ddot{\mathsf{main}}}$

4 Non-Cumulative Resource Analysis

In this section we present our approach to use the information obtained in Sec. 3 to infer the peak cost of the execution. The first part, Sec. 4.1, consists in performing a program-point resource analysis in which we are able to infer the resources acquired at the points of interest. In Sec. 4.2, we discard from the upper bound obtained before those resources which are not used simultaneously.

4.1 Program-Point Resource Analysis

Our goal is to distinguish within the upper bounds (UB) obtained by resource analysis the amount of resources acquired at a given program point. To do so, we rely on the notion of *cost center* (CC) [1]. Originally, CCs were introduced for the analysis of distributed systems, such that, each CC is a symbolic expression of the form $c(o)$ where o is a location identifier used to separate the cost of each distributed location. Essentially, the resource analysis assigns the cost of an instruction *inst* to the distributed location o by multiplying the cost due to the execution of the instruction, denoted $cost(inst)$ in a generic way, by the cost center of the location $c(o)$, i.e., $cost(inst){*}c(o)$. This way, the UBs that the analysis obtains are of the form $\sum c(o_i){*}C_i$, where each o_i is a location identifier and C_i is the total cost accumulated at this location.

Importantly, the notion of CC can be used in a more general way to define the granularity of a cost analyzer, i.e., the kind of separation that we want to observe in the UBs. In our concrete application, the expressions of the cost centers o_i will refer to the program points of interest. Thus, we are defining a resource

analyzer that provides the resource consumption at program point level, i.e., a *program point* resource analysis. In particular, we define a CC for each acquire instruction in the program. Thus, CCs are of the form $c(a_{pp})$ for each instruction pp:acquire(e). In essence, the analyzer every time that accounts for the cost of executing an acquire instruction multiplies such cost by its corresponding cost center. The amount of resources allocated at the instruction pp:acquire(e) is accumulated as an expression of the form $c(a_{pp})*\mathsf{nat}(e)$, where $\mathsf{nat}(e)$ is a function that returns e if $e>0$ and 0 otherwise. We wrap the expression e with nat because this way the analyzer treats it as a non-negative expression whose cost we want to maximize, and computes the worst case of such expression (technical details can be found in [2]). The cost analyzer computes an *upper bound* for the total cost of executing P as an expression of the form $\mathcal{U}_P(\bar{x})=\sum_{i=1}^{n} c(a_i)*C_i$, where C_i is a cost expression that bounds the resources allocated by the acquire instructions of the program. We omit the subscript in $\mathcal{U}$ when it is clear from the context. If one is interested in the amount of resources allocated by one particular acquire instruction a_{pp}, denoted $\mathcal{U}(\bar{x})|_{a_{pp}}$, we simply replace all $c(a_{pp'})$ with $pp \neq pp'$ by 0 and $c(a_{pp})$ by 1. We extend it to sets as $\mathcal{U}(\bar{x})|_S = \sum_{a_{pp}\in S} \mathcal{U}(\bar{x})|_{a_{pp}}$.

Example 5. The program point UB for the running example is:

$$\mathcal{U}(n,s) = \overbrace{c(a_1)*k_1}^{e_1} + \overbrace{c(a_2)*\mathsf{nat}(s)}^{e_2} + \underbrace{\overbrace{\mathsf{nat}(n) * (c(a_3)*k_2 + c(a_4)*\mathsf{nat}(s)) + c(a_5)*k_3}^{e_3} + \mathsf{nat}(n{+}2) * (c(a_3)*k_2 + c(a_4)*\mathsf{nat}(s{+}4)) + c(a_5)*k_3}_{e_4}$$

We have a CC for each acquire instruction in the program multiplied by the amount of resources allocated by the corresponding acquire. In the examples, we do not wrap constants in nat because constant values do not need to be maximized, e.g. in the subexpression e_1 which corresponds to the cost of L2. The subexpression e_2 corresponds to L4 where s units are allocated. Expression e_3 corresponds to the first call to q, where the loop iterates $\mathsf{nat}(n)$ times and consumes $c(a_3)*k_2$ ($L10$) and $c(a_4)*\mathsf{nat}(s)$ ($L11$) resources for each iteration, plus the final acquire at $L15$, which allocates $c(a_5)*k_3$ resources. The cost of the second call to q is captured by e_4, where the number of iterations is bounded by $\mathsf{nat}(n{+}2)$ and $\mathsf{nat}(s{+}4)$ resources are allocated. e_4 also includes the cost allocated at L15. Let us continue by using $\mathcal{U}(n,s)$ to compute the resources allocated at a particular location, e.g. a_4, denoted by $\mathcal{U}(n,s)|_{a_4}$. To do so, we replace $c(a_4)$ by 1 and the rest of $c(\_)$ by 0. Thus, $\mathcal{U}(n,s)|_{a_4} = \mathsf{nat}(n)*\mathsf{nat}(s)+\mathsf{nat}(n{+}2)*\mathsf{nat}(s{+}4)$. Similarly, given the set of program points $\{a_3, a_5\}$, we have $\mathcal{U}(n,s)|_{\{a_3,a_5\}} = \mathcal{U}(n,s)|_{\{a_3\}} + \mathcal{U}(n,s)|_{\{a_5\}} = \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3$.

4.2 Inference of Peak Cost

We can now put all pieces together. The SRA described in Sec. 3 allows us to infer the acquire instructions which could be allocated simultaneously. Such information is gathered in the set $\mathcal{H}$ of the SRA. In fact, the set $\mathcal{H}$ at the last program point of the program, namely $\ddot{\mathsf{main}}$, collects all possible states of the resource allocation during program execution. Using this set we define the notion of *peak cost* as the maximum of the UBs computed for each possible set in $\mathcal{H}_{\ddot{\mathsf{main}}}$.

Definition 2 (peak cost). *The* peak cost *of a program* $P(\overline{x})$*, denoted* $\widehat{\mathcal{P}}(\overline{x})$*, is defined as* $\widehat{\mathcal{P}}(\overline{x}) = max(\{\mathcal{U}(\overline{x})|_{\mathbb{H}} \mid \mathbb{H} \in \mathcal{H}_{\ddot{main}}\})$.

Intuitively, for each $\mathbb{H}$ in $\mathcal{H}_{\ddot{\mathsf{main}}}$, we compute its restricted UB, $\mathcal{U}(\overline{x})|_{\mathbb{H}}$, by removing from $\mathcal{U}(\overline{x})$ the cost due to acquire instructions that are not in $\mathbb{H}$, i.e., those acquire that were not active simultaneously with the elements in h.

Example 6. By using $\mathcal{H}_{\ddot{\mathsf{m}}} = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}, \{a_2, a_3, a_4, a_5\}\}$, the peak cost of m is the maximum of the expressions:

$$\begin{aligned}
\mathcal{U}(n,s)|_{\{a_1,a_3,a_4\}} &= k_1 + \mathsf{nat}(n)*(k_2 + \mathsf{nat}(s)) + \mathsf{nat}(n{+}2)*(k_2 + \mathsf{nat}(s{+}4))\\
\mathcal{U}(n,s)|_{\{a_1,a_3,a_5,a_2\}} &= k_1 + \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3\\
\mathcal{U}(n,s)|_{\{a_2,a_3,a_4,a_5\}} &= \mathsf{nat}(s) + \mathsf{nat}(n)*(k_2{+}\mathsf{nat}(s)) + k_3 + \mathsf{nat}(n{+}2)*(k_2{+}\mathsf{nat}(s{+}4)){+}k_3
\end{aligned}$$

Each UB expression over-approximates the value of R for the different states seen in Ex. 3 that could determine the concrete peak cost, namely $\mathcal{U}(n,s)|_{\{a_1,a_3,a_4\}}$ over-approximates the resource consumption at state S_8, $\mathcal{U}(n,s)|_{\{a_1,a_3,a_5,a_2\}}$ corresponds to S_{12}, and $\mathcal{U}(n,s)|_{\{a_2,a_3,a_4,a_5\}}$ bounds S_{16} and S_{18}.

Theorem 2 (soundness). $\mathcal{P}(\overline{x}) \leq \widehat{\mathcal{P}}(\overline{x})$.

5 Extensions of the Basic Framework

In this section we discuss several extensions to our basic framework. First, Sec. 5.1 discusses how context-sensitive analysis can improve the accuracy of the results. Sec. 5.2 describes an improvement for handling *transient* acquire instructions, i.e., those resources which are allocated and released repeatedly but only one of all allocations is in use at a time. Finally, Sec. 5.3 introduces the extension of the framework to handle several kinds of resources.

5.1 Context-Sensitivity

Establishing the granularity of the analysis at the level of program points may lead to a loss of precision. This is because the computation of the SRA and the resource analysis are not able to distinguish if an acquire instruction is executed multiple times from different contexts. As a consequence, all resource usage associated to a given a_{pp} is accumulated in a single CC.

Example 7. The set $\mathcal{H}_{\ddot{m}}$ computed in Ex. 4 includes a_4 in two different sets. The first set corresponds to the first call to q (L3), where s units are allocated, whereas the second set corresponds to the second call (L6), and where s+4 units are allocated. Observe that the SRA of m does not distinguish such situation as both executions of L11 are represented as a single program point a_4. The same occurs in the computation of the UBs. In Ex. 6 we have computed $\mathcal{U}(n,s)|a_4 = \mathsf{nat}(n)*\mathsf{nat}(s)+\mathsf{nat}(n{+}2)*\mathsf{nat}(s{+}4)$, which accounts for the resources acquired at L11. Note that $\mathcal{U}(n,s)|a_4$ does not separate the cost of the different calls to q.

Intuitively, this loss of precision can be detected by checking if the *call graph* of the program contains convergence nodes, i.e., methods that have more than one incoming edge because they are invoked from different contexts. In such case, we can use standard techniques for context-sensitive analysis [15], e.g., method replication. In particular, the program can be rewritten by creating a different copy of the method for each incoming edge. Method replication guarantees that the calling contexts are not merged unless they correspond to a method call within a loop (or transitively from a loop). In the latter case, we indeed need to merge them and obtain the worst-case cost of all iterations, as the underlying resource analysis [2] already does.

Example 8. As q is called at L3 and L6, the application of the context-sensitive replication builds up a program with two methods: q_1 (from the call at L3) and q_2 (from L6). In addition, the modified version of m, denoted m', calls q_1 at L3 and q_2 at L6. We use a_{31} (resp. a_{32}) to refer to the acquire at L10 for the replica q_1 (resp. q_2). The SRA for m' returns: $\mathcal{H}_{\ddot{\mathsf{m}}'} = \{\{a_1, a_{31}, a_{41}\}, \{a_1, a_{31}, a_{51}, a_2\}, \{a_{31}, a_{51}, a_2, a_{32}, a_{42}\}, \{a_{31}, a_{51}, a_2, a_{32}, a_{52}\}\}$ and $\mathcal{C}_{\ddot{\mathsf{m}}'} = \{a_2, a_{31}, a_{32}, a_{51}, a_{52}\}$. Observe that the set marked with ⊛ in Ex. 4 is now split in two different sets, which precisely capture the states S_{16} and S_{18} of Fig. 2. Moreover, we distinguish a_{41}, a_{42} and a_{51}, a_{52} that allow us to separate the different calls to q, which is crucial for accounting the peak cost more accurately. The UB for m' is:

$$\mathcal{U}_{\mathsf{m}'}(n,s) = c(a_1)*k_1 + c(a_2)*\mathsf{nat}(s) + \mathsf{nat}(n)*(c(a_{31})*k_2 + c(a_{41})*\mathsf{nat}(s)) + c(a_{51})*k_3 + \mathsf{nat}(n{+}2)*(c(a_{32})*k_2 + c(a_{42})*\mathsf{nat}(s{+}4)) + c(a_{52})*k_3$$

In contrast to $\mathcal{U}_m(n,s)|_{a_4}$, shown in Ex. 5, now we can compute $\mathcal{U}_{m'}(n,s)|_{a_{41}} = \mathsf{nat}(n)*\mathsf{nat}(s)$ and $\mathcal{U}_{m'}(n,s)|_{a_{42}} = \mathsf{nat}(n{+}2)*\mathsf{nat}(s{+}4)$. $\widehat{\mathcal{P}}_{m'}(n,s)$ is the maximum of:

$$\begin{aligned}
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{41}\}} &= k_1 + \mathsf{nat}(n)*(k_2 + \mathsf{nat}(s)) && [S_8]\\
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{51},a_2\}} &= k_1 + \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 && [S_{12}]\\
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{42}\}} &= \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*(k_2{+}\mathsf{nat}(s{+}4)) && [S_{16}]\\
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{52}\}} &= \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3 && [S_{18}]
\end{aligned}$$

To the right of the UB expressions above we show their corresponding state of Fig. 2. In contrast to Ex. 6, now we have a one-to-one correspondence, and thus $\widehat{\mathcal{P}}_{m'}(n,s)$ is more accurate than $\widehat{\mathcal{P}}_m(n,s)$ in Ex. 6.

5.2 Handling Transient Resource Allocations

A complementary optimization with that in Sec. 5.1 can be performed when resources are acquired and released multiple times along the execution of the program within loops (or recursion). We use the notion of *transient* acquire to refer to an acquire(e) instruction at a_{pp} that is executed and released repeatedly but in such a way that the resources allocated by different executions of a_{pp} never coexist. As the UBs of Sec. 4 are computed by multiplying the number of times that each acquire instruction is executed by the worst case cost of each execution, the fact that the allocations of a transient acquire do not coexist is not accurately captured by the UB.

Example 9. Let us focus on the acquire a_4 of the running example. Although a_4 is executed multiple times within the loop, each allocation does not escape from the corresponding iteration because it is released at L12. To the right of Fig. 2 we can see that states S_3, S_6, S_8, S_{15} and S_{16} include the cost allocated by a_4 only once (elements in dark grey). Thus, a_4 is a transient acquire. In spite of this, we compute $\mathcal{U}_{\mathsf{m}'}(n,s)|_{a_{41}}=\mathsf{nat}(n)*\mathsf{nat}(s)$, which accounts for the cost allocated at a_{41} as many times as a_{41} might be executed. Certainly, $\mathcal{U}_{\mathsf{m}'}(n,s)|_{a_{41}}$ is a sound but imprecise approximation for the cost allocated by a_{41}.

We can improve the accuracy of the UBs for a transient acquire a_{pp} by including its worst case cost only once. We start by identifying when a_{pp} is transient in the concrete setting. Intuitively, if a_{pp} is transient the resources allocated at a_{pp} do not leak. Thus, in the last state of the execution, S_n, no resource allocated at a_{pp} remains in H_n (see the semantics at Fig. 1).

Definition 3 (transient acquire). *Given a program P, an acquire instruction a_{pp} is* transient *if for every execution trace of P, $S_1 \rightsquigarrow \ldots \rightsquigarrow S_n$, $\langle \_, \_, a_{pp}, \_\rangle \notin H_n$.*

Example 10. In Fig. 2 we can see that a_1 and a_4 (shown in dark grey) are transient because their resources are always released at L5 and L12, resp.

In order to count the cost of a transient acquire only once, we use a particular instantiation of the cost analysis described in Sec. 4.1 to determine an UB on the number of times that such acquire might be executed. We use $\mathcal{U}^c$ to denote such UB which is computed by replacing the expression C_i (see Sec. 4.1) by 1 in the computation of $\mathcal{U}$. Assuming that $\mathcal{U}$ and $\mathcal{U}^c$ have been approximated by the same cost analyzer, we gain precision by obtaining the cost associated to a transient acquire instruction using its *singleton cost*.

Definition 4 (singleton cost). *Given a_{pp} we define its* singleton cost *as $\widetilde{\mathcal{U}}(\overline{x})|_{a_{pp}} = \mathcal{U}(\overline{x})|_{a_{pp}} / \mathcal{U}^c(\overline{x})|_{a_{pp}}$ if $\_{:}a_{pp} \notin \mathcal{C}_{\ddot{\mathsf{main}}}$ and $\widetilde{\mathcal{U}}(\overline{x})|_{a_{pp}} = \mathcal{U}(\overline{x})|_{a_{pp}}$, otherwise.*

Intuitively, when a_{pp} is transient, its singleton cost is obtained dividing the accumulated UB by the number of times that a_{pp} is executed. If it is not transient, we must keep the accumulated UB. According to Def. 3 and Th. 1(b), if $a_{pp} \notin \mathcal{C}_{\ddot{\mathsf{main}}}$, then a_{pp} is transient, and so we can perform the division. We use $\widetilde{\mathcal{P}}$ to refer to the peak cost obtained by using $\widetilde{\mathcal{U}}$ instead of $\mathcal{U}$. In general, given a set of a_{pp}, we use $\widetilde{\mathcal{U}}_{\mathsf{m}'}|_S$ to refer to the UBs computed using the singleton cost of each $a_{pp} \in S$.

Example 11. Let us continue with the context-sensitive replica of the running example, m'. We start by computing $\mathcal{U}^c_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(n)$ and $\mathcal{U}_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(n) * \mathsf{nat}(s)$. As we can see in Ex. 8, $a_{41}, a_{42} \notin \mathcal{C}_{\ddot{m}'}$, then $\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(s)$ which is the worst case of executing a_{41} only once. For a_{42} we have $\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{a_{42}} = \mathsf{nat}(s{+}4)$. Regarding the remaining acquire instructions, either they cannot be divided, or can be divided by 1. Thus, we have that $\widetilde{\mathcal{P}}_{\mathsf{m}'}(n,s)$ is the maximum of the following expressions:

$$\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{41}\}} = k_1 + \mathsf{nat}(n)*k_2 + \mathsf{nat}(s) \quad [S_8]$$
$$\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{51},a_2\}} = k_1 + \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 \quad [S_{12}]$$
$$\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{42}\}} = \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(s{+}4) \quad [S_{16}]$$
$$\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{52}\}} = \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3 \quad [S_{18}]$$

Theorem 3 (soundness). *Given a program $P(\overline{x})$ and its context-sensitive replica $P'(\overline{x})$, we have that $\mathcal{P}_P(\overline{x}) \leq \widetilde{\mathcal{P}}_{P'}(\overline{x})$.*

5.3 Handling Different Resources Simultaneously

Our goal is now to allow allocation of different types of resources in the program (e.g., we want to infer the heap space usage and the number of simultaneous connections to a database). To this purpose, we extend the instruction acquire(e) (see Sec. 2.1) with an additional parameter which determines the kind of resource to be allocated, i.e., acquire(res,e). Such extension does not require any modification to the semantics. We define the function $type(a_{pp})$ which returns the type of resource allocated at a_{pp}. Now, we extend Def. 1 to consider the resource of interest. We use $R_i(\mathsf{res})$ to refer to the following value $R_i(\mathsf{res}) = \sum\{r \mid \langle \_, \_, a_{pp}, r\rangle \in H_i \wedge type(a_{pp}) = \mathsf{res}\}$.

Definition 5 (concrete peak cost). *Given a resource res, the* peak cost *of an execution trace t of program $P(\overline{x}, \mathsf{res})$ is $\mathcal{P}(\overline{x}, \mathsf{res}) = max(\{R_i(\mathsf{res}) | S_i \in t\})$.*

Interestingly, such extension does not require any modification neither to the SRA of Sec. 3 nor to the program point resource analysis of Sec. 4. This is due to the fact that the analysis works at the level of program points and one program point can only allocate one particular type of resource. We define $\mathcal{R}(\mathsf{res})$ as the set of program points that allocate resources of type res, i.e., $\mathcal{R}(\mathsf{res})=\{a_{pp} \mid type(a_{pp})=\mathsf{res}\}$. Thus, we extend the notion of peak cost of Def. 2 with the type of resource, i.e., $\widehat{\mathcal{P}}(\overline{x}, \mathsf{res})=max(\{\mathcal{U}(\overline{x})|_{\mathbb{H}\cap\mathcal{R}(\mathsf{res})} \mid \mathbb{H} \in \mathcal{H}_{\mathsf{main}}\})$. Observe that the only difference with Def. 2 is in the intersection $\mathbb{H} \cap \mathcal{R}(\mathsf{res})$ which restricts the considered acquire when computing the UBs. One relevant aspect is that by computing the UB only once, we are able to obtain the peak cost for different types of resources by restricting the UB for each resource of interest. The extension of Th. 2 and Th. 3 to include a particular resource is straightforward.

Example 12. Let us modify the acquire instructions of the running example in Fig. 2 to add the resource to be allocated. Now we have that L2 is x = acquire(hd,k_1) and L11 is r = acquire(hd,w), where hd is a type of resource. We assume that L4, L10, L15 allocate a different type of resource, e.g. a resource of type mem. Then, using the context-sensitive replica of the program, we have that $\mathcal{R}(\mathsf{hd}) = \{a_1, a_{41}, a_{42}\}$, and $\mathcal{R}(\mathsf{mem}) = \{a_2, a_{31}, a_{32}, a_{51}, a_{52}\}$. Now, using the UB from Ex. 11, we have that $\widehat{\mathcal{P}}(n, s, \mathsf{hd})_{\mathsf{m}'}$ is the maximum of the expressions:

$$\begin{aligned}
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_1,a_{31},a_{41}\}\cap\mathcal{R}(\mathsf{hd})} &= k_1{+}\mathsf{nat}(s) && [S_8]\\
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_1,a_{31},a_{51},a_2\}\cap\mathcal{R}(\mathsf{hd})} &= k_1 && [S_{12}]\\
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_{31},a_{51},a_2,a_{32},a_{42}\}\cap\mathcal{R}(\mathsf{hd})} &= \mathsf{nat}(s{+}4) && [S_{16}]\\
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_{31},a_{51},a_2,a_{32},a_{52}\}\cap\mathcal{R}(\mathsf{hd})} &= 0 && [S_{18}]
\end{aligned}$$

6 Experimental evaluation

We have implemented a prototype peak cost analyzer for simple sequential programs that follow the syntax of Sec. 2.1, but that besides use a functional language to define data types (the use of functions does not require any conceptual

Benchmark	$\#_l$	$\#_e$	$\mathbf{T}_n$	$\mathbf{T}_c$	$\%_n$	$\%_c$	$\%_s$	$\%_{cn}$	$\%_{sn}$	$\%_{sc}$
BBuffer	105	3125	928.0	1072.0	4.9	35.7	43.9	32.1	40.6	15.7
MailServer	115	3375	958.0	1233.0	16.0	42.4	58.2	30.2	47.1	27.6
Chat	302	2500	584.0	580.0	69.9	69.9	92.9	0.0	74.8	74.8
DistHT	353	2500	685.0	2267.0	40.2	82.8	84.8	71.2	74.6	10.7
BookShop	353	4096	2219.0	2409.0	6.5	6.5	32.4	0.0	27.9	27.9
PeerToPeer	240	4096	5616.0	11860.0	0.4	8.8	11.4	8.5	11.1	3.0
					23.0	41.0	53.9	23.7	46.0	26.6

Table 1. Experimental Evaluation

modification to our basic analysis). This language corresponds to the sequential sublanguage of ABS [12], a language which besides has concurrency features that are ignored by our analyzer. To perform the experiments, our analyzer has been applied to some programs written in ABS: BBuffer, a bounded-buffer for communicating producers and consumers; MailServer, a client-server system; Chat, a chat application; DistHT, an implementation of a hash table; BookShop, a book shop application; and PeerToPeer, a peer-to-peer network.

The non-cumulative resource that we measure is the peak of the size of the stack of activation records. For each method executed, an activation record is created, and later removed when the method terminates. The size might depend on the arguments used in the call, as due to the use of functional data structures, when a method is invoked, the data structures (used as parameters) are passed and stored. This aspect is interesting because we can measure the peak size, not only due to activation records whose size is constant, but also measure the size of the data structures used in the invocations, and take them into account.

In order to evaluate our analysis we have obtained different UBs on the size of the stack of activation records and compared their precision. In particular, we have compared the UBs obtained by the resource analysis of [2] (a cumulative cost analyzer), our basic non-cumulative approach (Sec. 4.2), the context-sensitive extension of Sec. 5.1 and the UBs obtained by using the singleton cost of each acquire as described in Sec. 5.2. In order to obtain concrete values for the gains, we have evaluated the UB expressions for different combinations of the input arguments and computed the average. For a concrete input arguments $\overline{x}$, we compute the gain of $\widehat{\mathcal{P}}(\overline{x})$ w.r.t. $\mathcal{U}(\overline{x})$ using the formula $(1-\widehat{\mathcal{P}}(\overline{x})/\mathcal{U}(\overline{x}))*100$. In order to compute the sizes of the activation records of the methods, we have modified each method of the benchmarks by including in the beginning of the method one acquire and one release at the end of each method to free it. Let us illustrate it with an example, if we have a method Int m (Data d,Int i) {Int j=i+1}, we modify it to {x=acquire(1+1+d+1+1); Int j=i+1; release x;}. The addends of the expression 1+1+d+1+1 correspond to: the pointer to the activation record, the size of the returned value (1 unit), the size of the information received through d (d units), the size of i (1 unit), and the size of j (1 unit). The instruction release(x) releases all resources. Experiments have been performed on an Intel Core i5 (1.8GHz, 4GB RAM), running OSX 10.8.

Table 1 summarizes the results obtained. Columns $\#_l$ and $\#_e$ show, resp., the number of lines of code and the number of input argument combinations

evaluated. Columns $\mathbf{T}_n$, $\mathbf{T}_c$ show, resp., the time to perform the basic non-cumulative analysis and the context-sensitive non-cumulative analysis. Columns $\%_n$, $\%_c$, $\%_s$ show, resp., the gain of the non-cumulative resource analysis, its context-sensitive extension and the singleton cost extension w.r.t. the cumulative analysis. Column $\%_{cn}$ shows the gain of $\widehat{\mathcal{P}}$ applied to the context sensitive replica of the program w.r.t. its application to the original program. Columns $\%_{sn}$ and $\%_{sc}$ show, resp., the gain of $\widetilde{\mathcal{P}}$ w.r.t. $\widehat{\mathcal{P}}$, and w.r.t. $\widehat{\mathcal{P}}$ applied to the context sensitive replica of the program. The last row shows the average of the results. As regards analysis times, we argue that the time taken by the analyzer is reasonable and the context-sensitive approach although more expensive is feasible. As regards precision, we can observe that the gains obtained by the non-cumulative analyses are significant w.r.t. the cumulative resource analysis. As it can be expected, $\widetilde{\mathcal{P}}$ shows the best results with gains from 11% to 93%. The non-cumulative analysis and its context-sensitive version also present significant gains, on average 23% and 41% respectively. The improvement gained by applying non-cumulative analysis to the context-sensitive extension is also relevant, a gain of 23.7%. As resources are released in all methods, we achieve a significant improvement with $\widetilde{\mathcal{P}}$, from 46% to 26.6% on average. All in all, we argue that the experimental evaluation shows the accuracy of non-cumulative resource analysis and the precision gained with its extensions.

7 Conclusions and Related Work

To the best of our knowledge, this is the first generic framework to infer the peak of the resource consumption of sequential imperative programs. The crux of the framework is an analysis to infer the resources that might be used simultaneously along the execution. This analysis is formalized as a data-flow analysis over a finite domain of sets of resources. The inference is followed by a program-point resource analysis which defines the resource consumption at the level of the program points at which resources are acquired.

Previous work on non-cumulative cost analysis of sequential imperative programs has been focused on the particular resource of memory consumption with garbage collection, while our approach is generic on the kind of non-cumulative cost that one wants to measure. Our framework can be used to redefine previous analyses of heap space usage [5] into the standard cost analysis setting. Depending on the particular garbage collection strategy, the release instruction will be placed at one point or another. For instance, if one uses scope-based garbage collection, all release instructions are placed just before the method return instruction and our framework can be applied. If one wants to use a liveness-based garbage collection, then the liveness analysis determines where the release instructions should go, and our analysis is then applied. The important point to note is that these analyses [5] provided a solution based on the generation of non-standard cost relations specific to the problem of memory consumption. It thus cannot be generalized to other kind of non-cumulative resources. Several analyses around the RAML tool [11] also assume the existence of acquire and release instructions and the application of our framework to this setting is an

interesting topic for further research. The differences between amortized cost analysis and a standard cost analysis are discussed in [6,9]. Also, we want to study the recasting of [7] into our generic framework.

Recent work defines an analysis to infer the peak cost of distributed systems [3]. There are two fundamental differences with our work: (1) [3] is developed for cumulative resources, and the extension to non-cumulative resources is not studied there and (2) [3] considers a concurrent distributed language, while our focus is on sequential programs. There is nevertheless a similarity with our work in the elimination from the total cost of elements that do not happen simultaneously. However, in the case of [3] this information is gathered by a complex may-happen-in-parallel analysis [4] which infers the interleavings that may occur during the execution followed by a post-process in which a graph is built and its cliques are used to detect when several tasks can be executing concurrently. In our case, we are able to detect when resources are used simultaneously by means of a simpler analysis defined as a standard data-flow analysis on a finite domain. Besides, the upper bounds in [3] are not obtained by a program-point resource analysis but rather by a task-level resource analysis since in their case they want to obtain the resource consumption at the granularity of tasks rather than of program points. As in our case, the use of context sensitive analysis [15] can improve the accuracy of the results.

References

1. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *TACAS'14*, vol. 8413 of *LNCS*, pp 562–567. 2014.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP'07*, vol. 4421 of *LNCS*, pp 157–172. Springer, 2007.
3. E. Albert, J. Correas, and G. Román-Díez. Peak Cost Analysis of Distributed Systems. In *SAS'14*, vol. 8723 of *LNCS*, pp 18–33. Springer, 2014.
4. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *FORTE'12*, vol. 7273 of *LNCS*, pp 35–51. Springer, 2012.
5. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *ISMM'10*, pp 121-130. 2010.
6. D. E. Alonso-Blas and S. Genaim. On the Limits of the Classical Approach to Cost Analysis. In *SAS 2012*, vol. 7460 of *LNCS*, pp 405–421. Springer, 2012.
7. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM'08*, pp 141–150. ACM, 2008.
8. V. A. Braberman, D. Garbervetsky, S. Hym, and S. Yovine. Summary-based inference of quantitative bounds of live heap objects. *SCP*, 92:56–84, 2014.
9. A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS'14*, vol. 8858 of *LNCS*, pp 275–295. Springer, 2014.
10. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL'09*, pp 127–139. ACM, 2009.
11. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'13*, pp 185–197. ACM, 2003.

12. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO'10 (Revised Papers)*, vol. 6957 of *LNCS*, pp 142–164. Springer, 2012.
13. Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. of CAV'14*, volume 8559, pages 745–761. Springer, 2014.
14. P. W. Trinder, M. I. Cole, K. Hammond, H.W. Loidl, and G. Michaelson. Resource analyses for parallel and distributed coordination. *CCPE*, 25(3):309–348, 2013.
15. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

Appendix G

Article *Peak Cost Analysis of Distributed Systems*, [10]

Peak Cost Analysis of Distributed Systems

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

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

Abstract. We present a novel static analysis to infer the *peak cost* of distributed systems. The different locations of a distributed system communicate and coordinate their actions by posting tasks among them. Thus, the amount of work that each location has to perform can greatly vary along the execution depending on: (1) the amount of tasks posted to its queue, (2) their respective costs, and (3) the fact that they may be posted in parallel and thus be pending to execute *simultaneously*. The peak cost of a distributed location refers to the maximum cost that it needs to carry out along its execution. Inferring the peak cost is challenging because it increases and decreases along the execution, unlike the standard notion of *total cost* which is cumulative. Our key contribution is the novel notion of *quantified queue configuration* which captures the worst-case cost of the tasks that may be simultaneously pending to execute at each location along the execution. A prototype implementation demonstrates the accuracy and feasibility of the proposed peak cost analysis.

1 Introduction

Distributed systems are increasingly used in industrial processes and products, such as manufacturing plants, aircraft and vehicles. For example, many control systems are decentralized using a distributed architecture with different processing locations interconnected through buses or networks. The software in these systems typically consists of concurrent tasks which are statically allocated to specific locations for processing, and which exchange messages with other tasks at the same or at other locations to perform a collaborative work. A decentralized approach is often superior to traditional centralized control systems in performance, capability and robustness. Systems such as control systems are often critical: they have strict requirements with respect to timing, performance, and stability. A failure to meet these requirements may have catastrophic consequences. To verify that a given system is able to provide the required quality, an essential aspect is to accurately predict *worst-case costs*. We develop our analysis for a generic notion of cost that can be instantiated to the number of executed instructions (considered as the best abstraction of time for software), the amount of memory created, the number of tasks posted to each location, or any other *cost model* that assigns a non-negative cost to each instruction.

Existing cost analyses for distributed systems infer the *total* resource consumption [3] of each distributed location, e.g., the total number of instructions

that it needs to execute, the total amount of memory that it will need to allocate, or the total number of tasks that will be added to its queue. This is unfortunately a too pessimistic estimation of the amount of resources actually required in the real execution. An important observation is that the *peak* cost will depend on whether the tasks that the location has to execute are pending *simultaneously*. We aim at inferring such peak of the resource consumption which captures the maximum amount of resources that the location might require along any execution. In addition to its application to verification as described above, this information is crucial to dimensioning the distributed system: it will allow us to determine the size of each location *task queue*; the required size of the location's memory; and the processor execution speed required to execute the peak of instructions and provide a certain response time. It is also of great relevance in the context of software *virtualization* as used in cloud computing, as the peak cost allows estimating how much processing/storage capacity one needs to buy in the host machine, and thus can greatly reduce costs.

This paper presents, to the best of our knowledge, the first static analysis to infer the peak of the resource consumption of distributed systems, which takes into account the type and amount of tasks that the distributed locations can have in their queues simultaneously along any execution, to infer precise bounds on the peak cost. Our analysis works in three steps: (1) *Total cost analysis.* The analysis builds upon well-established analyses for total cost [9,3,18]. We assume that an underlying total cost analysis provides a *cost* for the tasks which measures their efficiency. (2) *Queues configurations.* The first contribution is the inference of the *abstract queue configuration* for each distributed component, which captures all possible configurations that its queue can take along the execution. A particular queue configuration is given as the sets of tasks that the location may have pending to execute at a moment of time. We rely on the information gathered by a *may-happen-in-parallel* analysis [7,1,11,5] to define the abstract queue configurations. (3) *Peak cost.* Our key contribution is the notion of *quantified queue configuration*, which over-approximates the peak cost of each distributed location. For a given queue configuration, its quantified configuration is computed by removing from the total cost inferred in (1) those tasks that do not belong to its configuration, as inferred in (2). The peak for the location is the maximum of the costs of all configurations that its queue can have.

We demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer of peak cost within the SACO system [2], a static analyzer for distributed concurrent programs. In preliminary experiments on some typical applications for distributed programs, the peak cost achieves gains up to 70% w.r.t. a total cost analysis. The tool can be used on-line from a web interface available at `http://costa.ls.fi.upm.es/web/saco`.

2 The Distributed Model

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered queue

$$\frac{t = tsk(tid, m, l, \langle x = \mathsf{newLoc}; s\rangle, c),\ \mathrm{fresh}(lid_1)\ ,\ l' = l[x \to lid_1]}{loc(lid, tid, \{t\}\cup\mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s, c+cost(\mathsf{newLoc}))\}\cup\mathcal{Q}) \| loc(lid_1, \bot, \{\})} \quad (\textsc{newLoc})$$

$$\frac{l(x) = lid_1,\ \mathrm{fresh}(tid_1),\ l_1{=}buildLocals(\bar{z}, m_1),\ l' = l[f \to tid_1]}{\begin{array}{c} loc(lid, tid, \{tsk(tid, m, l, \langle f{=}x!m_1(\bar{z}); s\rangle, c)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \mathcal{Q}') \rightsquigarrow \\ loc(lid, tid, \{tsk(tid, m, l', s, c + cost(f{=}x!m_1(\bar{z})))\}\cup\mathcal{Q}) \parallel \\ loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1), 0) \cup \mathcal{Q}'\}) \end{array}} \quad (\textsc{async})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle, c), l(f) = tid_1, tsk(tid_1, \_, \_, s_1, \_) \in \mathsf{Locs}, s_1 = \tau}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, s, c + cost(\mathsf{await}\ f?))\} \cup \mathcal{Q})} \quad (\textsc{await-t})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle, c), l(f) = tid_1, tsk(tid_1, \_, \_, s_1, \_) \in \mathsf{Locs}, s_1 \neq \tau}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle, c)\} \cup \mathcal{Q})} \quad (\textsc{await-f})$$

$$\frac{select(\mathcal{Q}){=}tid,\quad t{=}tsk(tid, \_, \_, s, c){\in}\mathcal{Q}, s{\neq}\tau}{loc(lid, \bot, \mathcal{Q}){\rightsquigarrow}loc(lid, tid, \mathcal{Q})} \quad (\textsc{select})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{return}; \rangle, c)}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \tau, c{+}cost(\mathsf{return}))\}\cup\mathcal{Q})} \quad (\textsc{return})$$

Fig. 1. (Summarized) Cost Semantics for Distributed Execution

of pending tasks. Initially all processors are idle. When an idle processor's task queue is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the queues of any processor, including its own, and synchronize with the completion of tasks. When a task completes or when it is awaiting for another task to terminate, its processor becomes idle again, chooses the next pending task, and so on.

2.1 Syntax

The number of distributed locations needs not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to an *object* and can be dynamically created using the instruction newLoc. The program is composed by a set of methods defined as $M{::=}T\ m(\bar{T}\ \bar{x})\{s\}$ where $s{::=}\ s; s \mid x{=}e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{return} \mid b{=}\mathsf{newLoc} \mid f{=}b!m(\bar{e}) \mid \mathsf{await}\ f?$. The notation $\bar{T}$ is used as a shorthand for $T_1, \ldots, T_n$, and similarly for other names. The special location identifier *this* denotes the current location. For the sake of generality, the syntax of expressions e and types T is left open. The semantics of future variables f and concurrency instructions is explained below.

2.2 Semantics

A *program state* has the form $loc_1 \| \ldots \| loc_n$, denoting the currently existing distributed locations. Each *location* is a term $loc(lid, tid, \mathcal{Q})$ where *lid* is the location identifier, *tid* is the identifier of the *active task* which holds the location's lock or $\bot$ if the lock is free, and $\mathcal{Q}$ is the set of tasks at the location. Only one task, which holds the location's *lock*, can be *active* (running) at this location. All

other tasks are *pending*, waiting to be executed, or *finished*, if they terminated and released the lock. Given a location, its set of *ready* tasks is composed by the tasks that are pending and the one that it is active at the location. A *task* is a term $tsk(tid, m, l, s, c)$ where tid is a unique task identifier, m is the name of the method executing in the task, l is a mapping from local variables to their values, s is the sequence of instructions to be executed or $s = \tau$ if the task has terminated, and c is a positive number which corresponds to the cost of the instructions executed in the task so far. The cost of executing an instruction i is represented in a generic way as $cost(i)$.

The execution of a program starts from a method m in an initial state S_0 with a single (initial) location of the form $S_0{=}loc(0, 0, \{tsk(0, \mathsf{m}, l, body(\mathsf{m}), 0)\})$. Here, l maps parameters to their initial values and local references to null (standard initialization), and $body(m)$ refers to the sequence of instructions in the method m. The execution proceeds from the initial state S_0 by selecting *nondeterministically* one of the locations and applying the semantic rules depicted in Fig. 1. The treatment of sequential instructions is standard and thus omitted. The operational semantics $\rightsquigarrow$ is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows. In NewLoc, an active task tid at location lid creates a location lid_1 which is introduced to the state with a free lock. Async spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is added to the queue of location lid_1. The case $lid{=}lid_1$ is analogous, the new task tid_1 is simply added to $\mathcal{Q}$ of lid. The future variable f allows synchronizing the execution of the current task with the termination of created task. The association of the future variable to the task is stored in the local variables table l'. In Await-t, the future variable we are awaiting for points to a finished task and await can be completed. The finished task t_1 is looked up at all locations in the current state (denoted by Locs). Otherwise, Await-f yields the lock so that any other task of the same location can take it. Rule Select returns a task that is not finished, and it obtains the lock of the location. Return releases the lock and it will never be taken again by that task. Consequently, that task is *finished* (marked by adding τ). For brevity, we omit the return instructions in the examples.

3 Peak Cost of Distributed Systems

The aim of this paper is to infer an *upper bound* on the *peak cost* for all locations of a distributed system. The peak cost refers to the maximum amount of resources that a given location might require along any execution. The overapproximation consists in computing the sum of the costs of all tasks that can be simultaneously ready in the location's queue. Importantly, as the number of ready tasks in the queue can increase and decrease along the execution, in order to define the notion of peak cost, we need to observe all intermediate states along the computation and take the maximum of their costs.

Example 1. Figure 2 shows to the left a method m that spawns several tasks at a location referenced from variable x (the middle code can be ignored by now). To

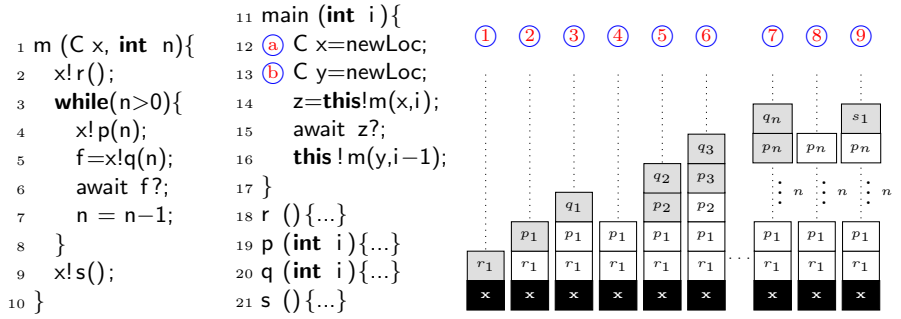


Fig. 2. Running example

the right of the figure, we depict the tasks that are ready in the queue of location x at different states of the execution of m. For instance, the state ① is obtained after invoking method r at line 2 (L2 for short). The first iteration of the while loop spawns a task p (state ②) and then invokes q (state ③). The important observation is that q is awaited at L6, and thus it is guaranteed to be finished at state ④. The same pattern is repeated in subsequent loop iterations (states ⑤ to ⑦). The last iteration of the loop, captured in state ⑦, accumulates all calls to p, and the last call to q. Observe that at most one instance of method q appears in the queue at any point during the execution of the program. Finally, state ⑧ represents the exit of the loop (L8) and ⑨ when method s is invoked at L9. The await at L6 ensures that methods q and s will not be queued simultaneously.

We start by formalizing the notion of peak cost in the concrete setting. Let us provide some notation. Given a state $S{\equiv}loc_1 \| \ldots \| loc_n$, we use $loc \in S$ to refer to a location in S. The set of ready tasks at a location lid at state S is defined as $ready\_tasks(S, lid) = \{tid \mid loc(lid, \_, \mathcal{Q}) \in S, tsk(tid, \_, \_, s, \_){\in}\mathcal{Q}, s{\neq}\tau\}$. Note that we exclude the tasks that are finished. Given a finite trace $t \equiv S_0 {\rightarrow} \ldots {\rightarrow} S_N$, we use $\mathcal{C}(lid, tid)$ to refer to the accumulated cost c in the final state S_N by the task $tsk(tid, \_, \_, \_, c){\in}\mathcal{Q}$ that executes at location $loc(lid, \_, \mathcal{Q}){\in}S_N$, and $\mathcal{C}(S_i, lid)$ to refer to the accumulated cost of all active tasks that are in the queue at state S_i for location lid: $\mathcal{C}(S_i, lid) = \sum_{tid \in ready\_tasks(S_i, lid)} \mathcal{C}(lid, tid)$. Now, the peak cost of location lid is defined as the *maximum* of the addition of the costs of the tasks that are simultaneously ready at the location at any state: $peak\_cost(t, lid) = \max(\{\mathcal{C}(S_i, lid) \mid S_i \in t\})$. Observe that the *cost* always refers to the cost of each task in the *final* state S_N. This way we are able to capture the cost that a location will need to carry out at each state S_i with $i \leq N$ in which we have a set of ready tasks in its queue but they have (possibly) not been executed yet.

Since execution is non-deterministic in the selection of tasks, given a program $P(x)$, multiple (possibly infinite) traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible traces for $P(\overline{x})$.

Definition 1 (peak cost). *The peak cost of a location with identifier lid in a program P on input values $\overline{x}$, denoted $\mathcal{P}(P(\overline{x}), lid)$, is defined as:*

$\mathcal{P}(P(\overline{x}), lid) = max(\{peak\_cost(t, lid) \mid t \in executions(P(\overline{x}))\})$

Example 2. Let us reason on the peak cost for the execution of m. We use $\ddot{m}$ to refer to the concrete cost of a task executing method m. We use subscripts $\ddot{m}_j$ to refer to the cost of the j-th task spawned executing method m. As the cost often depends on the parameters, in general, we have different costs $\ddot{m}_1$, $\ddot{m}_2$,... for the multiple executions of the same method. The queue of x in states ② and ④ accumulates the cost $\ddot{r}_1 + \ddot{p}_1$. At ⑥, it accumulates $\ddot{r}_1 + \ddot{p}_1 + \ddot{p}_2 + \ddot{p}_3 + \ddot{q}_3$. The peak cost corresponds to the maximum among all states. Note that it is unknown if the cost at ⑦ is larger than the cost at ③-⑤-⑥-.... This is because at each state we have a different instance of q running, and it can be that $\ddot{q}_1$ is larger than the whole cost of the next iterations. Only some states can be discarded (for instance ① and ② are subsumed by ③, and ⑧ by ⑨).

The above example reveals several important aspects that make the inference of the peak cost challenging: (1) We need to infer all possible queue configurations. This is because the peak cost is non-cumulative, and any state can require the maximum amount of resources and constitute the peak cost. This contrasts with the total cost in which we only need to observe the final state. (2) We need to track when tasks terminate their execution and eliminate them from the configuration (the await instructions will reveal this information). (3) We need to know how many instances of tasks we might have running and bound the cost of each instance, as they might not all have the same cost.

4 Basic Concepts: Points-to, Cost, and MHP Analyses

Our peak cost analysis builds upon well-established work on points-to analysis [14,13], total cost analysis [9,18,3] and may-happen-in-parallel (MHP) analysis [11,5]. As regards the points-to and may-happen-in-parallel analyses, this section only reviews the basic concepts which will be used later by our peak cost analysis. As for the total cost analysis, we need to tune several components of the analysis in order to produce the information that the peak cost analysis requires.

Points-to Analysis. Since locations can be dynamically created, we need an analysis that abstracts them into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. Points-to analysis [14,13,16] solves this problem. It infers the set of memory locations that a reference variable can *point-to*. Different abstractions can be used and our method is parametric on the chosen abstraction. Any points-to analysis that provides the following information with more or less accurate precision can be used (our implementation uses [13]): (1) $\mathcal{O}$, the set of abstract locations; (2) $\mathcal{M}(o)$, the set of methods executed in tasks at the abstract location $o \in \mathcal{O}$; (3) a function $pt(pp, v)$ which for a given program point pp and a variable v returns the set of abstract locations in $\mathcal{O}$ to which v may point to.

Example 3. Consider the main method shown in Fig. 2, which creates two new locations x at program point ⓐ (abstracted as o_1) and y at ⓑ (abstracted as o_2) and passes them as parameters in the calls to m (at L14, L16). By using the

points-to analysis we obtain the following relevant information, $\mathcal{O}$=$\{\epsilon, o_1, o_2\}$ where ϵ is the location executing main, $\mathcal{M}(o_1)$=$\{r, p, q, s\}$, $\mathcal{M}(o_2)$=$\{r, p, q, s\}$, $pt(L14, x)$=$\{o_1\}$ and $pt(L16, y)$=$\{o_2\}$. Observe that the abstract task executing p at location o_1 represents multiple instances of the tasks invoked at L4.

Cost Analysis. The notion of *cost center* (CC) is an artifact used to define the granularity of a cost analyzer. In [3], the proposal is to define a CC for each distributed location, i.e., CCs are of the form $c(o)$ where $o \in \mathcal{O}$. In essence, the analyzer every time that accounts for the cost of executing an instruction b at program point pp, it also checks at which locations it is executing. This information is provided by the points-to analysis as O_{pp}=$pt(pp, this)$. The cost of the instruction is accumulated in the CCs of all elements in O_{pp} as $\sum c(o)*cost(b), \forall o \in O_{pp}$, where $cost(b)$ expresses in an abstract way the cost of executing the instruction. If we are counting steps, then $cost(b) = 1$. If we measure memory, $cost(b)$ refers to memory created by b. Then, given a method $m(\bar{x})$, the cost analyzer computes an *upper bound* for the total cost of executing m of the form $\widehat{\mathcal{C}}_m(\bar{x})$=$\sum_{i=1}^{n} c(o_i)*C_i$, where o_i∈$\mathcal{O}$ and C_i is a cost expression that bounds the cost of the computation carried out by location o_i when executing m. We omit the subscript in $\widehat{\mathcal{C}}$ when it is clear from the context. Thus, CCs allow computing costs at the granularity level of the distributed locations. If one is interested in studying the computation performed by one particular location o_j, denoted $\widehat{\mathcal{C}}_m(\bar{x})|_{o_j}$, we simply replace all $c(o_i)$ with i≠j by 0 and $c(o_j)$ by 1. The use of CCs is of general applicability and different approaches to cost analysis (e.g., cost analysis based on recurrence equations [17], invariants [9] or type systems [10]) can trivially adopt this idea so as to extend their frameworks to a distributed setting. In principle, our method can work in combination with any analysis for total cost (except for the accuracy improvement in Sec. 5.3).

Example 4. By using the points-to information obtained in Ex. 3, a cost analyzer (we use in particular [2]) would obtain the following upper bounds on the cost distributed at the locations o_1 and o_2 (we ignore location ϵ in what follows as it is not relevant): $\widehat{\mathcal{C}}_{main}(i)$=$c(o_1)*\widehat{r_1} + c(o_1)*i*\widehat{p_1} + c(o_1)*i*\widehat{q_1} + c(o_1)*\widehat{s_1} + c(o_2)*\widehat{r_2} + c(o_2)*(i-1)*\widehat{p_2} + c(o_2)*(i-1)*\widehat{q_2} + c(o_2)*\widehat{s_2}$. There are two important observations: (1) the analyzer computes the *worst-case* cost $\widehat{p_1}$ for all instances of tasks spawned at L4 executing p at location o_1 (note that it is multiplied by the number of iterations of the loop "i"); (2) the upper bound at location o_2 for the tasks executing p is $\widehat{p_2}$, and it is different from $\widehat{p_1}$ as the invocation to m at L16 has different initial parameters. By replacing $c(o_1)$ by 1 we obtain the cost executed at the location identified by o_1, that is, $\widehat{\mathcal{C}}_{main}|_{o_1}$=$\widehat{r_1} + i*\widehat{p_1} + i*\widehat{q_1} + \widehat{s_1}$.

Context-Sensitive Task-level Cost Centers. Our only modification to the total cost analysis consists in using *context-sensitive task-level* granularity by means of appropriate CCs. Let us first focus on the task-level aspect. We want to distinguish the cost of the tasks executing at the different locations. We define task-level cost centers, $\overline{\mathcal{T}}$, as the set $\{o{:}m \in \mathcal{O} \times \mathcal{M} \mid o \in pt(pp, this) \wedge pp \in m\}$, which contains all methods combined with all location names that can execute

them. In the example, $\overline{\mathcal{T}}=\{\epsilon{:}m, o_1{:}r, o_1{:}p, o_1{:}q,\ o_1{:}s, o_2{:}r, o_2{:}p, o_2{:}q, o_2{:}s\}$. Now, the analyzer every time that accounts for the cost of executing an instruction *inst*, it checks at which location it is executing (e.g., o) and to which method it belongs (e.g., m), and it accumulates $c(o{:}m){*}cost(b)$. Thus, it is straightforward to modify an existing cost analyzer to include task-level cost centers. The context-sensitive aspect refers to the fact that the whole cost analysis can be made context-sensitive by considering the calling context when analyzing the tasks [15]. As usual, the context is the *chain of call sites* (i.e., the program point in which the task is spawned and those of its ancestor calling methods). The length of the chains is up to a maximum k which is a fixed parameter of the analysis. For instance, for k=2, we distinguish 14:4:p the task executing p from the first invocation to m at L14 and 16:4:p the one arising from L16. Their associated CCs are then o_1:14:4:p and o_2:16:4:p. In the formalization, we assume that the context (call site chain) is part of the method name m and thus we write CCs simply as $c(o{:}m)$. Then, given an entry method $p(\bar{x})$, the cost analyzer will compute a *context-sensitive task-level upper bound* for the cost of executing p of the form $\widehat{\mathcal{C}}_p(\bar{x})=\sum_{i=1}^{n} c(o_i{:}m_i){*}C_i$, where $o_i{:}m_i \in \overline{\mathcal{T}}$, and C_i is a cost expression that bounds the cost of the computation carried out by location o_i executing method m_i, where m_i contains the calling context. The notation $\widehat{\mathcal{C}}_p(\bar{x})|_{o:m}$ is used to obtain the cost associated with $c(o{:}m)$ within $\widehat{\mathcal{C}}_p(\bar{x})$, i.e., the one obtained by setting to zero all $c(o'{:}m')$ with $o'{\neq}o$ or $m'{\neq}m$ and to one $c(o{:}m)$.

Example 5. For the method main shown in Fig. 2, the cost expression obtained by using task-level CCs and k=0 (i.e., making it context insensitive) is the following: $\widehat{\mathcal{C}}(i){=}c(o_1{:}r){*}\widehat{r_1}{+}c(o_1{:}p){*}i{*}\widehat{p_1}{+}c(o_1{:}q){*}i{*}\widehat{q_1}{+}c(o_1{:}s){*}\widehat{s_1}{+}c(o_2{:}r){*}\widehat{r_2}{+}c(o_2{:}p){*}(i{-}1){*}\widehat{p_2}{+}c(o_2{:}q){*}(i{-}1){*}\widehat{q_2}{+}c(o_2{:}s){*}\widehat{s_2}$. To obtain the cost carried out by o_1 when executing q, we replace $c(o_1{:}q)$ by 1 and the remaining CCs by 0, resulting in $\widehat{\mathcal{C}}(i)|_{o_1:q}{=}i{*}\widehat{q_1}$. For $k{>}0$, we simply add the call site sequences in the CCs, e.g., $c(o_1$:14:4:p$)$.

May-Happen-in-Parallel Analysis. We use a MHP analysis [11,5] as a black box and assume the same context and object-sensitivity as in the cost analysis. We require that it provides us: (1) The set of MHP pairs, denoted $\tilde{\mathcal{E}}_P$, of the form $(o_1{:}p_1, o_2{:}p_2)$ which indicates that program point p_1 running at location o_1 and program point p_2 running at location o_2 might execute in parallel. (2) A function $nact(o{:}m)$ that returns 1 if only one instance of m can be active at location o or ∞ if we might have more than one ([5] provides both 1 and 2).

Example 6. An MHP analysis [5] infers for the main method in Fig. 2, among others, the following set of MHP pairs at location o_1, $\{(o_1{:}18, o_1{:}19), (o_1{:}18, o_1{:}20), (o_1{:}18, o_1{:}21), (o_1{:}19, o_1{:}20), (o_1{:}19, o_1{:}21)\}$. In essence, each pair is capturing that the corresponding methods might happen in parallel, e.g., $(o_1{:}18, o_1{:}19)$ implies that methods r and p might happen in parallel. The MHP analysis learns information from the await to capture that only one instance of q can be active at location o_1, thus $nact(o_1{:}q){=}1$. On the contrary, the number of active calls to p is greater than 1, then $nact(o_1{:}p){=}\infty$.

5 Peak Cost Analysis

In this section we present our framework to infer the peak cost. It consists of two main steps: we first infer in Sec. 5.1 the configurations that the (abstract) location queue can feature (we use the MHP information in this step); and in a second step, we compute in Sec. 5.2 the cost associated with each possible queue configuration (we use the total cost in this step). Finally, we discuss in Sec. 5.3 an important extension of the basic framework that can increase its accuracy.

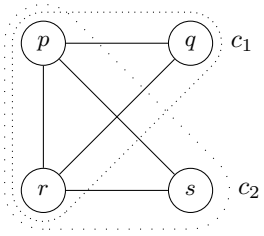


Fig. 3. $\mathcal{G}_t(o_1)$ for Fig 2

5.1 Inference of Queue Configurations

Our goal now is to infer, for each abstract location in the program, all its *non-quantified* configurations, i.e., the sets of method names that can be executing in tasks that are simultaneously ready in the location's queue at some state in the execution. Configurations are non-quantified because we ignore how many instances of a method can be pending in the queue and their costs.

Definition 2 (tasks queue graph). *Given a program P, an abstract location $o \in \mathcal{O}$ and the results of the MHP analysis $\tilde{\mathcal{E}}_P$, the* tasks queue graph *for o $\mathcal{G}_t(o)=\langle V_t, E_t\rangle$ is an undirected graph where $V_t = \mathcal{M}(o)$ and $E_t = \{(m_1, m_2) \mid (p_1, p_2) \in \tilde{\mathcal{E}}_P, p_1 \in m_1, p_2 \in m_2, m_1 \neq m_2\}$.*

It can be observed in the above definition that when we have two program points that may-happen-in-parallel in the location's queue, then we add an edge between the methods to which those points belong.

Example 7. By using the MHP information for location o_1 in Ex. 6, we obtain the tasks queue graph $\mathcal{G}_t(o_1)$ shown in Fig. 3 with the following set of edges $\{(r,p),(r,q),(r,s),(p,s),(p,q)\}$ (dotted lines will be explained later).

The tasks queue graph allows us to know the sets of methods that may be ready in the queue simultaneously. This is because, if two methods might be queued at the same time, there must be an edge between them in the tasks queue graph. It is then possible to detect the subsets of methods that can be queued at the same: those that are connected with edges between every two nodes that represent such subset, i.e., they form a *clique*. Since we aim at finding the maximum number of tasks that can be queued simultaneously, we need to compute the *maximal cliques* in the graph. Formally, given an undirected graph $\mathcal{G}=\langle V, E\rangle$, a *maximal clique* is a set of nodes $C \subseteq V$ such that every two nodes in C are connected by an edge, and there is no other node in $V \backslash C$ connected to every node in C.

Example 8. For $\mathcal{G}_t(o_1)$ in Fig. 3, we have two maximal cliques: $c_1 = \{p, q, r\}$ and $c_2 = \{p, r, s\}$, which capture the states ⑦ and ⑨ of the queue of o_1 (see Fig. 2). Observe that the maximal cliques subsume other states that contain subsets of a maximal clique. For instance, states ①-⑥ are subsumed by c_1.

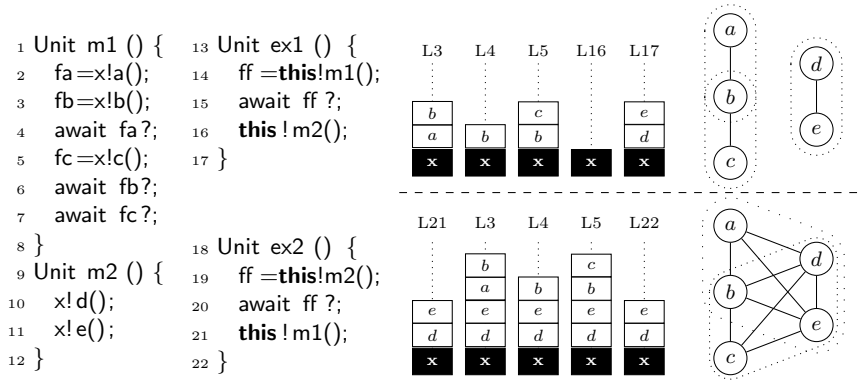


Fig. 4. Queue Configurations Example

Definition 3 (queue configuration). *Given a location o, we define its queue configuration, denoted by $\mathcal{K}(o)$, as the set of maximal cliques in $\mathcal{G}_t(o)$.*

Therefore, a queue configuration is a set of sets, namely each element in $\mathcal{K}(o)$ is a set of method names which capture a possible configuration of the queue. Clearly, all possible (maximal) configurations must be considered in order to obtain an over-approximation of the peak cost.

Example 9. Let us see a more sophisticated example for queue configurations. Consider the methods in Fig. 4 which have two distinct entry methods, ex1 and ex2. They both invoke method m1, which spawns tasks a, b and c. m1 guarantees that a, b and c are completed when it finishes. Besides, we know that b and c might run in parallel, while the await instruction in L4 ensures that a and c cannot happen in parallel. Method m2 spawns tasks d and e and does not await for their termination. We show in the middle of Fig. 4 the different configurations of the queue of x (at the program points marked on top) when we execute ex1 (above) and ex2 (below). Such configurations provide a graphical view of the results of the MHP analysis (which basically contains pairs for each two elements in the different queue states). In the queue of ex1, we can observe that the await instructions at the end of m1 guarantee that the queue is empty before launching m2 (see queue at L16). To the right of the queue we show the resulting tasks queue graph for ex1 obtained by using the MHP pairs which correspond to the queues showed in the figure. Then, we have $\mathcal{K}(x)$=$\{\{a,b\}$, $\{b,c\},\{d,e\}\}$. Note that these cliques capture the states of the queue at L3, L5 and L17, respectively. As regards ex2, the difference is that m2 is spawned before m1. Despite the await at L21, m2 is not awaiting for the termination of d nor e, thus at L21 the queue might contain d and e. As for m1, we have a similar behaviour than before, but now we have to accumulate also d and e along the execution of m1. The resulting tasks queue graph is showed to the right. It can be observed that it is densely connected, and now $\mathcal{K}(x)$=$\{\{d,e,a,b\},\{d,e,b,c\}\}$. Such cliques correspond to the states of the queue at L3 and L5, respectively.

5.2 Inference of Quantified Queue Configurations

In order to quantify queue configurations and obtain the peak cost, we need to over-approximate: (1) the number of instances that we might have running simultaneously for each task, (2) the worst-case cost of such instances. The main observation is that the upper bounds on the total cost in Sec. 4 already contain both types of information. This is because the cost attached to the CC $c(o{:}m)$ accounts for the accumulation of the resource consumption of *all* tasks running method m at location o. We therefore can safely use $\widehat{\mathcal{C}}(\overline{x})|_{o:m}$ as upper bound of the cost associated with the execution of method m at location o.

Example 10. According to Ex. 5, the costs accumulated in the CCs of $o_1{:}q$ and $o_1{:}p$ are $\widehat{\mathcal{C}}(i)|_{o_1:p} = i * \widehat{p}$ and $\widehat{\mathcal{C}}(i)|_{o_1:q} = i * \widehat{q}$. Note that $o_1{:}q$ is accumulating the cost of *all* calls to `q`, as the fact that there is at most one active call to `q` is not taken into account by the total cost analysis. This is certainly a sound but imprecise over-approximation that will be improved in Sec. 5.3.

The key idea to infer the *quantified queue configuration*, or simply *peak cost*, of each location is to compute the total cost for each element in the set $\mathcal{K}(o)$ and stay with the maximum of all of them. Given an abstract location o and a clique $k \in \mathcal{K}(o)$, we have that $\widehat{\mathcal{C}}(\overline{x})|_k = \sum_{m \in k} \widehat{\mathcal{C}}(\overline{x})|_{o:m}$ is the cost for the tasks in k.

Definition 4. *Given a program $P(\overline{x})$ and an abstract location o, the peak cost for o, denoted $\widehat{\mathcal{P}}(P(\overline{x}), o)$, is defined as $\widehat{\mathcal{P}}(P(\overline{x}), o) = \max(\{\widehat{\mathcal{C}}(\overline{x})|_k \mid k \in \mathcal{K}(o)\})$.*

Intuitively, as the elements of $\mathcal{K}$ capture all possible configurations that the queue can take, it is sound to take the maximum cost among them.

Example 11. Following Ex. 8, the quantified queue configuration, that gives the peak cost, accumulates the cost of all nodes in the two cliques, $\widehat{\mathcal{C}}(i)|_{c_1}{=}\widehat{r}{+}i{*}\widehat{p}{+}i{*}\widehat{q}$ and $\widehat{\mathcal{C}}(i)|_{c_2}{=}\widehat{r}{+}i{*}\widehat{p}{+}\widehat{s}$. The maximum between both expressions captures the peak cost for o_1, $\widehat{\mathcal{P}}(main(i), o_1) = max(\ \{\widehat{r}{+}i{*}\widehat{p}{+}i{*}\widehat{q}, \widehat{r}{+}i{*}\widehat{p}{+}\widehat{s}\})$.

The following theorem states the soundness of our approach.

Theorem 1 (soundness). *Given a program P with arguments $\overline{x}$, a concrete location lid, and its abstraction o, we have that $\mathcal{P}(P(\overline{x}), lid) \leq \widehat{\mathcal{P}}(P(\overline{x}), o)$.*

5.3 Number of Tasks Instances

As mentioned above, the basic approach has a weakness. From the queue configuration, we might know that there is at most one task running method m at location o. However, if we use $\widehat{\mathcal{C}}(\overline{x})|_{o:m}$, we are accounting for the cost of all tasks running method m at o. We can improve the accuracy as follows. First, we use an instantiation of the cost analysis in Sec. 4 to determine how many instances of tasks running m at o we might have. This can be done by defining function *cost* in Sec. 4 as follows: $cost(inst) = 1$ if $inst$ is the entry instruction to a method, and 0 otherwise. We denote by $\widehat{\mathcal{C}^c}(\overline{x})$ the upper bound obtained using such cost model that counts the number of tasks spawned along the execution, and $\widehat{\mathcal{C}^c}(\overline{x})|_{o:m}$ the number of tasks executing m at location o.

Example 12. The expression that bounds the number of calls from main is $\widehat{\mathcal{C}}^c(i)$= $c(o_1{:}r)+i{*}c(o_1{:}p)+i{*}c(o_1{:}q)+c(o_1{:}s)+c(o_2{:}r)+(i{-}1){*}c(o_2{:}p)+(i{-}1){*}c(o_2{:}q)+c(o_2{:}s)$. It can be seen that CCs are the same as the ones used in Ex. 5. The difference is that when inferring the number of calls we do not account for the cost of each method but rather count 1. Then, $\widehat{\mathcal{C}}^c(i)|_{o_1:q} = i$ and $\widehat{\mathcal{C}}^c(i)|_{o_2:q} = i{-}1$.

Let us assume that the same cost analyzer has been used to approximate $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{C}}^c$, and that the analysis assumed the worst-case cost of m for *all* instances of m. Then, we can gain precision by obtaining the cost as $\widetilde{\mathcal{C}}(\overline{x})|_{o:m} = \widehat{\mathcal{C}}(\overline{x})|_{o:m}/\widehat{\mathcal{C}}^c(\overline{x})|_{o:m}$ if $nact(o{:}m) = 1$ and $\widetilde{\mathcal{C}}(\overline{x})|_{o:m} = \widehat{\mathcal{C}}(\overline{x})|_{o:m}$, otherwise. Intuitively, when the MHP analysis tells us that there is at most one instance of m (by means of *nact*) and, under the above assumptions, the division is achieving the desired effect of leaving the cost of one instance only.

Example 13. As we have seen in Ex. 6, the MHP analysis infers $nact(o_1{:}p) = \infty$ and $nact(o_1{:}q) = 1$. Thus, by the definition of $\widetilde{\mathcal{C}}$, the cost for p is $\widetilde{\mathcal{C}}(i)|_{o_1:p} = i * \widehat{p}$ (the same obtained in Ex. 10). However, for q we can divide the cost accumulated by all invocations to q by the number of calls to q, $\widetilde{\mathcal{C}}(i)|_{o_1:q} = i * \widehat{q}/i = \widehat{q}$.

Unfortunately, it is not always sound to make such division. The problem is that the cost accumulated in a CC for a method m might correspond to the cost of executions of m from different calling contexts that do not necessarily have the same worst-case cost. If we divide the expression $\widehat{\mathcal{C}}(\overline{x})|_{o:m}$ by the number of instances, we are taking the average of the costs, and this is not a sound upper bound of the peak cost, as the following example illustrates.

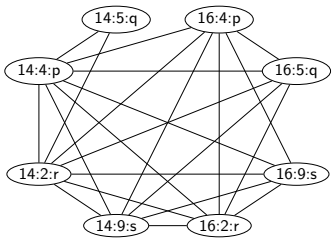


Fig. 5. Queue Config. for Fig. 2

Example 14. Consider a method main′ which is as main of Fig. 2 except that we replace L16 by this!m(x, i − 1), i.e., while main uses two different locations, x and y, in main′ we only use x. Such modification affects the precision because it merges o_1 and o_2 in a single queue, o_1. Now, in main′, s, launched by the first call to m, might run in parallel with q, spawned in the second call to m. Therefore, in Fig. 3 a new edge that connects q and s appears, and consequently, the new queue configuration contains all methods in just one clique $\{p, q, r, s\}$. Moreover, CCs $o_1{:}q$ with $o_2{:}q$ are merged in a single CC $o_1{:}q$. For main′, the cost of $\widehat{q}$ is $\widehat{\mathcal{C}}(i)|_{o:q}{=}i{*}\widehat{q}+(i{-}1){*}\widehat{q}$, and the number of calls is $\widehat{\mathcal{C}}^c(i)|_{o:q}{=}i+(i{-}1)$. Assume that the cost of q is $\widehat{q} = n * 5$ which is a function on the parameter n. The worst-case cost for $\widehat{q}$ depends on the calling context: in the context at L14, we have $\widehat{q} = i{*}5$ while in L16, we have $\widehat{q} = (i{-}1){*}5$. Then, the cost that we obtain for main′ is $\widehat{\mathcal{C}}(i)|_{o:q}{=}i{*}i{*}5+(i{-}1){*}((i{-}1){*}5)$. The division of $\widehat{\mathcal{C}}(i)$ by $\widehat{\mathcal{C}}^c(i)$ is not sound because it computes the average cost of all calls to q, rather than the peak.

Importantly, we can determine when the above division is sound in a static way. The information we are seeking is within the *call graph* for the program: (1) If there are not convergence nodes in the call graph (i.e., the call graph is a tree), then it is guaranteed that we do not have invocations to the same method from different contexts. In this case, if there are multiple invocations, it is because we are invoking m from the same context within a loop. Typically, automated cost analyzers assume the same worst-case cost for all loop iterations and, in such case, it is sound to make the division. Note that if the total cost analysis infers a different cost for each loop iteration, the accuracy improvement in this section cannot be applied; (2) If there are convergence nodes, then we need to ensure that the context-sensitive analysis distinguishes the calls that arise from different points, i.e., we have different CCs for them. This can be ensured if the length of the chains of call sites used in the context by the analysis, denoted k, is larger than k_d, the depth of the subgraph of the call graph whose root is the first convergence node encountered. Note that, in the presence of recursive methods, we will not be able to apply this accuracy improvement since the depth is unbounded. Theorem 1 holds for $\widetilde{\mathcal{C}}$ if the context considered by the analysis is greater than k_d.

Theorem 2. *Let $\widetilde{\mathcal{P}}(P(\overline{x}),o)$ be the peak cost computed using $\widetilde{\mathcal{C}}$. We have that $\mathcal{P}(P(\overline{x}),lid)\leq\widetilde{\mathcal{P}}(P(\overline{x}),o)$ if $k>k_d$, where k is the length of the context used.*

Example 15. Let us continue with main′ of Ex. 14. Assuming that p, q, r and s do not make any further call, the call graph has m as convergence node, and thus k_d=1. Therefore, we apply the context-sensitive analysis with k=2. The context-sensitive analysis distinguishes 14:4:p, 16:4:p, and, in q, 14:5:q and 16:5:q. The queue configuration is showed in Fig. 5. In contrast to Ex. 14 we have three different cliques, $\mathcal{K}(o_1)$={{14:4:p, 14:2:r, 14:5:q}, {14:4:p, 14:2:r, 14:9:s, 16:4:p, 16:2:r, 16:5:q}, {14:4:p, 14:2:r, 14:9:s, 16:4:p, 16:2:r, 16:9:s}}, which capture more precisely the queue states (e.g., we know that 16:5:q cannot be in the queue with 16:9:s but it might be with 14:9:s). Besides, we have two different CCs for q, 14:5:q and 16:5:q, which allow us to safely apply the division as to obtain the cost of a single instance of q for the two different contexts. We obtain $\widehat{\mathcal{C}}(i)|_{14:5:q}=i*i*5$ and $\widehat{\mathcal{C}}(i)|_{16:5:q}=(i-1)*((i-1)*5)$, and for the number of calls, $\widehat{\mathcal{C}}^c(i)|_{14:5:q}=i$ and $\widehat{\mathcal{C}}^c(i)|_{16:5:q}=i-1$. Using such expressions we compute $\widetilde{\mathcal{C}}(i)|_{14:5:q} = i*5$ and $\widetilde{\mathcal{C}}(i)|_{16:5:q} = (i-1)*5$ which are sound and precise over-approximations for the cost due to calls to q.

6 Experimental evaluation

We have implemented our analysis in SACO [2] and applied it to some typical examples of distributed systems: BBuffer, a bounded-buffer for communicating producers and consumers; MailS, a client-server distributed system; Chat, a chat application; DistHT, a distributed hash table; and P2P, a peer-to-peer network. Experiments have been performed on an Intel Core i5 (1.8GHz, 4GB RAM), running OSX 10.8. Table 1 summarizes the results obtained. Columns Bench.

				Context Insensitive					Context Sensitive				
Bench.	loc	$\#_{\mathsf{c}}$	T	$\#_{\mathsf{q}}$	$\%_{\overline{q}}$	$\%_m$	$\%_M$	$\%_{\widehat{\mathcal{P}}}$	$\#'_{\mathsf{q}}$	$\%'_{\overline{q}}$	$\%'_m$	$\%'_M$	$\%_{\widetilde{\mathcal{P}}}$
BBuffer	107	6	2.0	9	66.7%	50.0%	100%	78.1%	10	52.2%	17.6%	100%	31.6%
MailS	97	6	2.8	8	75.1%	71.6%	100%	81.7%	8	73.3%	71.6%	100%	81.7%
DistHT	150	4	2.5	8	69.4%	53.7%	100%	88.0%	8	69.4%	46.4%	100%	88.0%
Chat	328	10	2.4	16	66.0%	50.0%	100%	90.8%	16	66.0%	7.5%	100%	90.8%
P2P	259	9	28.0	26	52.9%	91.1%	100%	97.3%	32	32.3%	44.6%	100%	64.7%
Mean					66.0%	62.3%	100%	87.1%		58.6%	37.46%	100%	71.3%

Table 1. Experimental results (times in seconds)

and loc show, resp., the name and the number of program lines. Column $\#_{\mathsf{c}}$ shows the number of locations identified by the analysis. Columns T and $\#_{\mathsf{q}}$ show, resp., the time to perform the analysis and the number of cliques.

We aim at comparing the gain of using peak cost analysis w.r.t. total cost. Such gain is obtained by evaluating the expression that divides the peak cost by the total cost for 15 different values of the input parameters, and computing the average. The gain is computed at the level of locations, by comparing the peak cost for the location with the total cost for such location in all columns except in $\%_{\overline{q}}$, where we show the average gain at the level of cliques. Columns $\%_m$ and $\%_M$ show, resp., the greatest and smallest gain among all locations. Column $\%_{\widehat{\mathcal{P}}}$ shows the average gain weighted by the cost associated with each location (locations with higher resource consumption have greater weight). Columns $\#'_{\mathsf{q}}$, $\%'_{\overline{q}}$, $\%'_m$, $\%'_M$, and $\%_{\widetilde{\mathcal{P}}}$ contain the same information for the context-sensitive analysis. As we do not have yet an implementation of the context-sensitive analysis, we have replicated those methods that are called from different contexts. DistHT and Chat do not need replication. The last row shows the arithmetic mean of all results.

We can observe in the table that the precision gained by considering all possible queue states ($\%_{\overline{q}}$ and $\%'_{\overline{q}}$) is significant. In the context-insensitive analysis, it ranges from a gain of 53% to 75% (on average 66%). Such value is improved in the context sensitive analysis, resulting in an average gain of 58.6%. This indicates that the cliques capture accurately the cost accumulated in the different states of the locations' queues. The gain of the context sensitive analysis is justified by the larger number of cliques ($\#'_{\mathsf{q}}$) in BBuffer and P2P. The maximal gains showed in columns $\%_m$ (and $\%'_m$) indicate that the accuracy can be improved on average 62.3% (and 37.46%). The minimal gains in $\%_M$ and $\%'_M$ are always 100%, i.e., no gain. This means that in all benchmarks we have at least one state that accumulates the cost of all methods executed at its location (typically because await is never used). Columns $\%_{\widehat{\mathcal{P}}}$ and $\%_{\widetilde{\mathcal{P}}}$ show, in BBuffer and P2P, that $\widetilde{\mathcal{P}}$ significantly outperforms $\widehat{\mathcal{P}}$. Such improvement is achieved by a more precise configuration graph that contains more cliques, and by the division on the number of calls in methods that require a significant part of the resource consumption. However, in MailS, $\widetilde{\mathcal{P}}$ does not improve the precision of $\widehat{\mathcal{P}}$. This is because the methods that contain one active instance are not part of the cliques that lead to the peak

cost of the location. Despite of the NP-completeness of the clique problem, the time spent performing the clique computation is irrelevant in comparison with the time taken by the upper bound computation (less than 50ms for all benchmarks). All in all, we argue that our experiments demonstrate the accuracy of the peak cost analysis, even in its context insensitive version, with respect to the total cost analysis.

7 Conclusions, Related and Future Work

To the best of our knowledge, our work constitutes the first analysis framework for peak cost of distributed systems. This is an essential problem in the context of distributed systems. It is of great help to dimension the distributed system in terms of processing requirements, and queue sizes. Besides, it paves the way to the accurate prediction of response times of distributed locations. The task-level analysis in [4] is developed for a specific cost model that infers the peak of tasks that a location can have. There are several important differences with our work: (1) we are generic in the notion of cost and our framework can be instantiated to measure different types of cost, among them the task-level; (2) the distributed model that we consider is more expressive as it allows concurrent behaviours within each location (by means of instruction await), while [4] assumes a simpler asynchronous language in which tasks are run to completion; (3) the analysis requires the generation of non-standard recurrence equations, while our analysis benefits from the upper bounds obtained using standard recurrence equations for total cost, without requiring any modification. Indeed, the analysis in [4] could be reformulated in our framework using the MHP analysis of [11,12].

Also, the peak heap consumption in the presence of garbage collection is a non cumulative type of resource. The analysis in [6] presents a sophisticated framework for inferring the peak heap consumption by assuming different garbage collection models. As before, in contrast to ours, the analysis is based on generating non-standard equations and for a specific type of resource. In this case, the differences are even more notable as the language in [6] is sequential. Analysis and verification techniques of concurrent programs seek finite representations of the program traces which avoid the exponential explosion in the number of traces (see [8] and its references). In this sense, our queue configurations are a coarse representation of the traces. As future work, we plan to further improve the accuracy of our analysis by splitting tasks into fragments according to the processor release points within the task. Intuitively, if a task contains an await instruction we would divide into the code before the await and the code after. This way, we do not need to accumulate the cost of the whole task if only the fragment after the await has been queued.

Acknowledgments. 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

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPOPP*, pages 183–193. ACM, 2007.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
3. 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.
4. E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *Proc. of LCTES'11*, pages 21–30. ACM Press, 2011.
5. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
6. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Garbage Collected Languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.
7. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *LCPC'05*, volume 4339 of *LNCS*, pages 152–169. Springer, 2005.
8. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142. ACM, 2013.
9. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
10. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.
11. J. K. Lee and J. Palsberg. Featherweight x10: a core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, 2010.
12. J. K. Lee, J. Palsberg, and R. Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.
13. 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.
14. M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proc. of POPL'97*, pages 1–14, Paris, France, January 1997. ACM.
15. 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.
16. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
17. B. Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.
18. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

Appendix H

Article *Parallel Cost Analysis of Distributed Systems*, [7]

Parallel Cost Analysis of Distributed Systems *

Elvira Albert[1], Jesús Correas[1], Einar Broch Johnsen[2], Guillermo Román-Díez[3]

[1] DSIC, Complutense University of Madrid, Spain
[2] Dept. of Informatics, University of Oslo, Norway
[3] DLSIIS, Technical University of Madrid, Spain

Abstract. We present a novel static analysis to infer the *parallel cost* of distributed systems. Parallel cost differs from the standard notion of *serial cost* by exploiting the truly concurrent execution model of distributed processing to capture the cost of synchronized tasks executing in parallel. It is challenging to analyze parallel cost because one needs to soundly infer the parallelism between tasks while accounting for waiting and idle processor times at the different locations. Our analysis works in three phases: (1) It first performs a *block-level* analysis to estimate the serial costs of the blocks between synchronization points in the program; (2) Next, it constructs a *distributed flow graph* (DFG) to capture the parallelism, the waiting and idle times at the locations of the distributed system; Finally, (3) the parallel cost can be obtained as the path of maximal cost in the DFG. A prototype implementation demonstrates the accuracy and feasibility of the proposed analysis.

1 Introduction

Welcome to the age of distributed and multicore computing, in which software needs to cater for massively parallel execution. Looking beyond parallelism between independent tasks, *regular parallelism* involves tasks which are mutually dependent [17]: synchronization and communication are becoming major bottlenecks for the efficiency of distributed software. This paper is based on a model of computation which separates the asynchronous spawning of new tasks to different locations, from the synchronization between these tasks. The extent to which the software succeeds in exploiting the potential parallelism of the distributed locations depends on its synchronization patterns: synchronization points between dynamically generated parallel tasks restrict concurrency.

This paper introduces a novel static analysis to study the efficiency of computations in this setting, by approximating how synchronization between blocks of serial execution influences parallel cost. The analysis builds upon well-established static cost analyses for serial execution [2,8,21]. We assume that a serial cost analysis returns a "cost" for the serial blocks which measures their efficiency. Traditionally, the metrics used in cost analysis [19] is based on counting the

* This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.

number of execution steps, because this cost model appears as the best abstraction of time for software. Our parallel cost analysis could also be used in combination with worst-case execution time (WCET) analysis [1] by assuming that the cost of the serial blocks is given by a WCET analysis.

Previous work on cost analysis of distributed systems [2] accumulates costs from different locations, but ignores the parallelism of the distributed execution model. This paper presents, to the best of our knowledge, the first static analysis to infer the parallel cost of distributed systems which takes into account the parallel execution of code across the locations of the distributed system, to infer more accurate bounds on the parallel cost. Our analysis works in the following steps, which are the main contributions of the paper:

1. *Block-level cost analysis of serial execution.* We extend an existing cost analysis framework for the serial execution of distributed programs in order to infer information at the granularity of synchronization points.
2. *Distributed flow graph (DFG).* We define the notion of DFG, which allows us to represent all possible (equivalence classes of) paths that the execution of the distributed program can take.
3. *Path Expressions.* The problem of finding the parallel cost of executing the program boils down to finding the path of maximal cost in the DFG. Paths in the DFG are computed by means of the single-source path expression problem [18], which finds regular expressions that represent all paths.
4. *Parallel cost with concurrent tasks.* We leverage the previous two steps to the concurrent setting by handling tasks whose execution might suspend and interleave with the execution of other tasks at the same location.

We demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer of parallel cost within the SACO system, a static analyzer for distributed concurrent programs. Preliminary experiments on some typical applications for distributed programs achieve gains up to 29% w.r.t. a serial cost analysis. The tool can be used online from a web interface available at `http://costa.ls.fi.upm.es/web/parallel`.

2 The Model of Distributed Programs

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered buffer of pending tasks. Initially all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the buffer of any processor, including its own, and synchronize with the reception of tasks (synchronization will be presented later in Sec. 6). When a task completes, its processor becomes idle again, chooses the next pending task, and so on.

2.1 Syntax

The number of distributed locations need not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to an *object* and

$$(\textsc{newLoc})\ \frac{\text{fresh}(lid'),\ l' = l[x \to lid']}{\begin{array}{c} loc(lid, tid, \{tsk(tid, m, l, \langle x = \mathsf{newLoc}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow \\ loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid', \bot, \{\}) \end{array}}$$

$$(\textsc{async})\ \frac{l(x) = lid_1,\ \text{fresh}(tid_1),\ l_1 = buildLocals(\bar{z}, m_1)}{\begin{array}{c} loc(lid, tid, \{tsk(tid, m, l, \langle x.m_1(\bar{z}); s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow \\ loc(lid, tid, \{tsk(tid, m, l, s)\} \cup \mathcal{Q}) \| loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1))\}) \end{array}}$$

$$(\textsc{select})\ \frac{select(\mathcal{Q}) = tid,\ t = tsk(tid, \_, \_, s) \in \mathcal{Q}, s \neq \epsilon(v)}{loc(lid, \bot, \mathcal{Q}) \rightsquigarrow loc(lid, tid, \mathcal{Q})} \qquad (\textsc{return})\ \frac{v = l(x)}{\begin{array}{c} loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{return}\ x; \rangle)\} \cup \mathcal{Q}) \rightsquigarrow \\ loc(lid, \bot, \{tsk(tid, m, l, \epsilon(v))\} \cup \mathcal{Q}) \end{array}}$$

Fig. 1. Summarized Semantics for Distributed Execution

can be dynamically created using the instruction newLoc. The program consists of a set of methods of the form $M ::= T\ m(\overline{T\ x})\{s\}$. Statements s take the form $s ::= s; s \mid x = e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{return}\ x \mid x = \mathsf{newLoc} \mid x.m(\bar{z})$, where e is an expression, x, z are variables and m is a method name. The notation $\bar{z}$ is used as a shorthand for $z_1, \ldots, z_n$, and similarly for other names. The special location identifier *this* denotes the current location. For the sake of generality, the syntax of expressions e and types T is left open.

2.2 Semantics

A *program state* S has the form $loc_1 \| \ldots \| loc_n$, denoting the currently existing distributed locations. Each *location* is a term $loc(lid, tid, \mathcal{Q})$ where lid is the location identifier, tid the identifier of the *active task* which holds the location's lock or $\bot$ if the lock is free, and $\mathcal{Q}$ the set of tasks at the location. Only the task which holds the location's *lock* can be *active* (running) at this location. All other tasks are *pending*, waiting to be executed, or *finished*, if they have terminated and released the lock. A *task* is a term $tsk(tid, m, l, s)$ where tid is a unique task identifier, m the name of the method executing in the task, l a mapping from local variables to their values, and s the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available.

The execution of a program starts from a method m, in an initial state with an initial location with identifier 0 executing task 0 of the form $S_0 = loc(0, 0, \{tsk(0, \mathsf{m}, l, body(\mathsf{m}))\})$. Here, l maps parameters to their initial values and local references to null (standard initialization), and $body(\mathsf{m})$ refers to the sequence of instructions in the method m. The execution proceeds from S_0 by evaluating *in parallel* the distributed locations. The transition $\to$ denotes a parallel transition W in which we perform an evaluation step $\rightsquigarrow$ (as defined in Fig. 1) at every distributed location loc_i with $i = 1, \ldots, n$, i.e., $W \equiv loc_1 \| \ldots \| loc_n \to loc'_1 \| \ldots \| loc'_m$. If a location is idle and its queue is empty, the evaluation simply returns the same location state. Due to the dynamic creation of distributed locations, we have that $m \geq n$.

The transition relation $\rightsquigarrow$ in Fig. 1 defines the evaluation at each distributed location. The treatment of sequential instructions is standard and thus omitted. In NewLoc, an active task *tid* at location *lid* creates a location lid' with a free

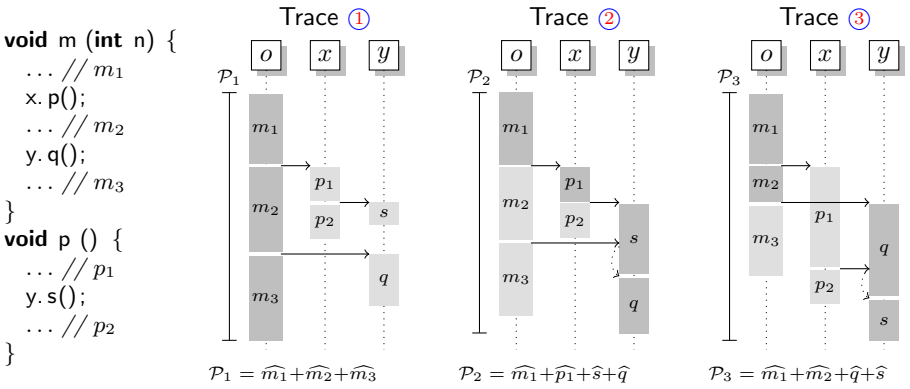


Fig. 2. Motivating example

lock, which extends the program state. This explains that $m \geq n$. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 in a singleton queue for the location lid_1 (which may be lid). We here elide the technicalities of remote queue insertion in the parallel transition step, which basically merges locations with the same identifier by taking the union of the queues. Rule SELECT returns a task that is not finished, and it obtains the lock of the location. When RETURN is executed, the return value is stored in v. In addition, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding instruction $\epsilon(v)$).

3 Parallel Cost of Distributed Systems

The aim of this paper is to infer an *upper bound* which is an over-approximation of the *parallel cost* of executing a distributed system. Given a parallel transition $W \equiv loc_1 \| \ldots \| loc_n \rightarrow loc'_1 \| \ldots \| loc'_m$, we denote by $\mathcal{P}(W)$ the parallel cost of the transition W. If we are interested in counting the number of executed transitions, then $\mathcal{P}(W) = 1$. If we know the time taken by the transitions, $\mathcal{P}(W)$ refers to the time taken to evaluate all locations. Thus, if two instructions execute in parallel, the parallel cost only accumulates the largest of their times. For simplicity, we assume that all locations execute one instruction in one cost unit. Otherwise, it must be taken into account by the cost analysis of the serial cost (see Sec. 8). Given a trace $t \equiv S_o \rightarrow \ldots \rightarrow S_{n+1}$ of the parallel execution, we define $\mathcal{P}(t) = \sum_{i=0}^{n} \mathcal{P}(W_i)$, where $W_i \equiv S_i \rightarrow S_{i+1}$. Since execution is non-deterministic in the selection of tasks, given a program $P(x)$, multiple (possibly infinite) traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible traces for $P(\overline{x})$.

Definition 1 (Parallel cost). *The parallel cost of a program P on input values $\overline{x}$, denoted $\mathcal{P}(P(\overline{x}))$, is defined as $max(\{\mathcal{P}(t) | t \in executions(P(\overline{x}))\})$.*

Example 1. Fig. 2 (left) shows a simple method m that spawns two tasks by calling p and q at locations x and y, resp. In turn, p spawns a task by calling s at location y. This program only features distributed execution, concurrent

behaviours within the locations are ignored for now. In the sequel we denote by $\widehat{m}$ the cost of block m. $\widehat{m_1}$, $\widehat{m_2}$ and $\widehat{m_3}$ denote, resp., the cost from the beginning of m to the call x.p(), the cost between x.p() and y.q(), and the remaining cost of m. $\widehat{p_1}$ and $\widehat{p_2}$ are analogous. Let us assume that the block m_1 contains a loop that performs n iterations (where n is equal to the value of input parameter n if it is positive and otherwise n is 0) and at each iteration it executes 10 instructions, thus $\widehat{m_1}$=$10 * n$. Let us assume that block m_2 contains a loop that divides the value of n by 2 and that it performs at most $log_2(n+1)$ iterations. Assume that at each iteration it executes 20 instructions, thus $\widehat{m_2}$=$20 * log_2(n+1)$. These expressions can be obtained by cost analyzers of serial execution [2]. It is not crucial for the contents of this paper to know how these expressions are obtained, nor what the cost expressions are for the other blocks and methods. Thus, in the sequel, we simply refer to them in an abstract way as $\widehat{m_1}$, $\widehat{m_2}$, $\widehat{p_1}$, $\widehat{p_2}$ etc. ■

The notion of parallel cost $\mathcal{P}$ corresponds to the cost consumed between the first instruction executed by the program at the initial location and the last instruction executed at any location by taking into account the parallel execution of instructions and idle times at the different locations.

Example 2. Fig. 2 (right) shows three possible traces of the execution of this example (more traces are feasible). Below the traces, the expressions $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ show the parallel cost for each trace. The main observation here is that the parallel cost varies depending on the duration of the tasks. It will be the worst (maximum) value of such expressions, that is, $\mathcal{P}$=$max(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \dots)$. In ② p_1 is shorter than m_2, and s executes before q. In ③, q is scheduled before s, resulting in different parallel cost expressions. In ①, the processor of location y becomes idle after executing s and must wait for task q to arrive. ■

In the general case, the inference of parallel cost is complicated because: (1) It is unknown if the processor is available when we spawn a task, as this depends on the duration of the tasks that were already in the queue; e.g., when task q is spawned we do not know if the processor is idle (trace ①) or if it is taken (trace ②). Thus, all scenarios must be considered; (2) Locations can be dynamically created, and tasks can be dynamically spawned among the different locations (e.g., from location o we spawn tasks at two other locations). Besides, tasks can be spawned in a circular way; e.g., task s could make a call back to location x; (3) Tasks can be spawned inside loops, we might even have non-terminating loops that create an unbounded number of tasks. The analysis must approximate (upper bounds on) the number of tasks that the locations might have in their queues. These points make the static inference of parallel cost a challenging problem that, to the best of our knowledge, has not been previously addressed. Existing frameworks for the cost analysis of distributed systems [3,2] rely on a *serial* notion of cost, i.e., the resulting cost accumulates the cost executed by all locations created by the program execution. Thus, we obtain a serial cost that simply adds the costs of all methods: $\widehat{m_1}$+$\widehat{m_2}$+$\widehat{m_3}$+$\widehat{p_1}$+$\widehat{p_2}$+$\widehat{q}$+$\widehat{s}$.

4 Block-level Cost Analysis of Serial Execution

The first phase of our method is to perform a *block-level* cost analysis of *serial* execution. This is a simple extension of an existing analysis in order to provide costs at the level of the blocks in which the program is partitioned, between synchronization points. In previous work, other extensions have been performed that use costs at the level of specific program points [4] or at the level of complete tasks [3], but the partitioning required by our parallel cost analysis is different. Later, we need to be able to cancel out the cost associated to blocks whose execution occurs in parallel with other blocks that have larger cost. The key notion of the extension is *block-level cost centers*, as defined below.

Block Partitioning. The need to partition the code into blocks will be clear when presenting the second phase of the analysis. Essentially, the subsequent analysis needs to have cost information for the following sets of blocks: $\mathcal{B}_{\mathsf{init}}$, the set of entry blocks for the methods; $\mathcal{B}_{\mathsf{exit}}$, the set of exit blocks for the methods, and $\mathcal{B}_{\mathsf{call}}$, the set of blocks ending with an asynchronous call. Besides these blocks, the standard partitioning of methods into blocks used to build the control flow graph (CFG) for the method is performed (e.g., conditional statement and loops introduce blocks for evaluating the conditions, edges to the continuations, etc.). We use $\mathcal{B}$ to refer to all block identifiers in the program. Given a block identifier b, $pred(b)$ is the set of blocks from which there are outgoing edges to block b in the CFG. Function $pred$ can also be applied to sets of blocks. We write $pp \in b$ (resp. $i \in b$) to denote that the program point pp (resp. instruction i) belongs to the block b.

Example 3. In Fig. 2, the traces show the partitioning in blocks for the methods m, p, q and s. Note that some of the blocks belong to multiple sets as defined above, namely $\mathcal{B}_{\mathsf{init}} = \{m_1, p_1, s, q\}$, $\mathcal{B}_{\mathsf{exit}}=\{m_3, p_2, s, q\}$, $\mathcal{B}_{\mathsf{call}}=\{m_1, m_2, p_1\}$. For instance, m_1 is both an entry and a call block, and s, as it is not partitioned, is both an entry and exit block. ∎

Points-to Analysis. Since locations can be dynamically created, we need an analysis that abstracts them into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. Points-to analysis [2,13,14] solves this problem. It infers the set of memory locations which a reference variable can *point-to*. Different abstractions can be used and our method is parametric on the chosen abstraction. Any points-to analysis that provides the following information with more or less accurate precision can be used (our implementation uses [2,13]): (1) $\mathcal{O}$, the set of abstract locations; (2) $\mathcal{M}$, the set of abstract tasks of the form $o.m$ where $o \in \mathcal{O}$ and m is a method name; (3) a function $pt(pp, v)$ which for a given program point pp and a variable v returns the set of abstract locations in $\mathcal{O}$ to which v may point to.

Example 4. In Fig. 2 we have three different locations, which are pointed to by variables o, x, y. For simplicity, we will use the variable name in italics to refer to the abstract location inferred by the points-to analysis. Thus, $\mathcal{O}=\{o, x, y\}$.

The abstract tasks spawned in the program are $\mathcal{M}$=$\{o.m, x.p, y.s, y.q\}$. In this example, the points-to abstraction is very simple. However, in general, locations can be reassigned, passed in parameters, have multiple aliases, etc., and it is fundamental to keep track of points-to information in an accurate way. ∎

Cost Centers. The notion of cost center is an artifact used to define the granularity of a cost analyzer. In [2], the proposal is to define a cost center for each distributed component; i.e., cost centers are of the form $c(o)$ where $o \in \mathcal{O}$ and $c(\_)$ is the artifact used in the cost expressions to attribute the cost to the different components. Every time the analyzer accounts for the cost of executing an instruction $inst$ at program point pp, it also checks at which location the instruction is executing. This information is provided by the points-to analysis as $O_{pp} = pt(pp, this)$. The cost of the instruction is accumulated in the cost centers of all elements in O_{pp} as $\sum c(o) * cost(inst), \forall o \in O_{pp}$, where $cost(inst)$ expresses in an abstract way the cost of executing the instruction. If we are counting steps, then $cost(inst) = 1$. If we measure time, $cost(inst)$ refers to the time to execute $inst$. Then, given a method $m(\bar{x})$, the cost analyzer will compute an upper bound for the serial cost of executing m of the form $\mathcal{S}_m(\bar{x}) = \sum_{i=1}^{n} c(o_i) * C_i$, where $o_i \in \mathcal{O}$ and C_i is a cost expression that bounds the cost of the computation carried out by location o_i when executing m. Thus, cost centers allow computing costs at the granularity level of the distributed components. If one is interested in studying the computation performed by one particular component o_j, we simply replace all $c(o_i)$ with $i \neq j$ by 0 and $c(o_j)$ by 1. The idea of using cost centers in an analysis is of general applicability and the different approaches to cost analysis (e.g., cost analysis based on recurrence equations [19], invariants [8], or type systems [9]) can trivially adopt this idea in order to extend their frameworks to a distributed setting. This is the only assumption that we make about the cost analyzer. Thus, we argue that our method can work in combination with any cost analysis for serial execution.

Example 5. For the code in Fig. 2, we have three cost centers for the three locations that accumulate the costs of the blocks they execute; i.e., we have $\mathcal{S}_m(n) = c(o) * \widehat{m_1} + c(o) * \widehat{m_2} + c(o) * \widehat{m_3} + c(x) * \widehat{p_1} + c(x) * \widehat{p_2} + c(y) * \widehat{s} + c(y) * \widehat{q}$. ∎

Block-level Cost Centers. In this paper, we need *block-level* granularity in the analysis. This can be captured in terms of block-level cost centers $\overline{\mathcal{B}}$ which contain all blocks combined with all location names where they can be executed. Thus, $\overline{\mathcal{B}}$ is defined as the set $\{o{:}b \in \mathcal{O} \times \mathcal{B} \mid o \in pt(pp, this) \wedge pp \in b\}$. We define $\overline{\mathcal{B}}_{\mathsf{init}}$ and $\overline{\mathcal{B}}_{\mathsf{exit}}$ analogously. In the motivating example, $\overline{\mathcal{B}} = \{o{:}m_1, o{:}m_2, o{:}m_3, x{:}p_1, x{:}p_2, y{:}q,\ y{:}s\}$. Every time the analyzer accounts for the cost of executing an instruction $inst$, it checks at which location $inst$ is executing (e.g., o) and to which block it belongs (e.g., b), and accumulates $c(o{:}b) * cost(inst)$. It is straightforward to modify an existing cost analyzer to include block-level cost centers. Given a method $m(\bar{x})$, the cost analyzer now computes a *block-level upper bound* for the cost of executing m. This upper bound is of the form $\mathcal{S}_m(\bar{x}) = \sum_{i=1}^{n} c(o_i{:}b_i) * C_i$, where $o_i{:}b_i \in \overline{\mathcal{B}}$, and C_i is a

cost expression that bounds the cost of the computation carried out by location o_i while executing block b_i. Observe that b_i need not be a block of m because we can have transitive calls from m to other methods; the cost of executing these calls accumulates in $\mathcal{S}_m$. The notation $\mathcal{S}_m(\bar{x})|_{o:b}$ is used to express the cost associated to $c(o{:}b)$ within the cost expression $\mathcal{S}_m(\bar{x})$, i.e., the cost obtained by setting all $c(o'{:}b')$ to 0 (for $o' \neq o$ or $b' \neq b$) and setting $c(o{:}b)$ to 1. Given a set of cost centers $N = \{o_0{:}b_0, \ldots, o_k{:}b_k\}$, we let $\mathcal{S}_m(\bar{x})|_N$ refer to the cost obtained by setting to one the cost centers $c(o_i{:}b_i)$ such that $o_i{:}b_i \in N$. We omit m in $\mathcal{S}_m(\bar{x})|_N$ when it is clear from the context.

Example 6. The cost of the program using the blocks in $\mathcal{B}$ as cost centers, is $\mathcal{S}_m(n){=}c(o{:}m_1){*}\widehat{m_1}{+}c(o{:}m_2){*}\widehat{m_2}{+}c(o{:}m_3){*}\widehat{m_3}{+}c(x{:}p_1){*}\widehat{p_1}{+}c(x{:}p_2){*}\widehat{p_2}{+}c(y{:}s){*}\widehat{s}{+}$ $c(y{:}q){*}\widehat{q}$. We can obtain the cost for block $o{:}m_2$ as $\mathcal{S}_m(n)|_{o:m_2} = \widehat{m_2}$. With the serial cost assumed in Sec. 3, we have $\mathcal{S}_m(n)|_{o:m_2} = 20 * log_2(n+1)$. ∎

5 Parallel Cost Analysis

This section presents our method to infer the cost of executing the distributed system by taking advantage of the fact that certain blocks of code must execute in parallel, thus we only need to account for the largest cost among them.

5.1 Distributed Flow Graph

The *distributed flow graph* (DFG), introduced below, aims at capturing the different flows of execution that the program can perform. According to the distributed model of Sec. 2, when the processor is released, any pending task of the same location could start executing. We use an existing *may-happen-in-parallel* (MHP) analysis [5,12] to approximate the tasks that could start their execution when the processor is released. This analysis infers pairs of program points (x, y) whose execution *might* happen in parallel. The soundness of the analysis guarantees that if (x, y) is not an MHP pair then there are no instances of the methods to which x or y belong whose program points x and y can run in parallel. The MHP analysis can rely on a points-to analysis in exactly the same way as our overall analysis does. Hence, we can assume that MHP pairs are of the form $(x{:}p_1, y{:}p_2)$ where x and y refer to the locations in which they execute. We use the notation $x{:}b_1 \parallel y{:}b_2$, where b_1 and b_2 are blocks, to denote that the program points of $x{:}b_1$ and $y{:}b_2$ might happen in parallel, and, $x{:}b_1 \nparallel y{:}b_2$ to indicate that they cannot happen in parallel.

Example 7. The MHP analysis of the example shown in Fig. 2 returns that $y{:}s \parallel y{:}q$, indicating that s and q might happen in parallel at location y. In addition, as we only have one instance of m and p, the MHP guarantees that $o{:}m_1 \nparallel o{:}m_3$ and $x{:}p_1 \nparallel x{:}p_2$. ∎

The nodes in the DFG are the cost centers which the analysis in Sec. 4 has inferred. The edges represent the control flow in the sequential execution (drawn with normal arrows) and all possible orderings of tasks in the location's queues (drawn with dashed arrows). We use the MHP analysis results to eliminate the dashed arrows that correspond to unfeasible orderings of execution.

Definition 2 (Distributed flow graph). *Given a program P, its block-level cost centers $\overline{\mathcal{B}}$, and its points-to analysis results provided by function pt, we define its* distributed flow graph *as a directed graph $\mathcal{G} = \langle V, E \rangle$ with a set of vertices $V = \overline{\mathcal{B}}$ and a set of edges $E = E_1 \cup E_2 \cup E_3$ defined as follows:*

$$E_1 = \{o{:}b_1 \rightarrow o{:}b_2 \mid b_1 \rightarrow b_2 \text{ exists in } CFG\}$$
$$E_2 = \{o_1{:}b_1 \rightarrow o_2{:}\mathsf{m}_{init} \mid b_1 \in \mathcal{B}_{call}, pp : \mathsf{x}.\mathsf{m}() \in b_1, o_2 \in pt(pp, \mathsf{x})\}$$
$$E_3 = \{o{:}b_1 \dashrightarrow o{:}b_2 \mid b_1 \in \mathcal{B}_{exit}, b_2 \in \mathcal{B}_{init}, o{:}b_1 \parallel o{:}b_2\}$$

Here, E_1 is the set of edges that exist in the CFG, but using the points-to information in $\overline{\mathcal{B}}$ in order to find out at which locations the blocks are executed. E_2 joins each block that contains a method invocation with the initial block $\mathsf{m}_{\mathsf{init}}$ of the invoked method. Again, points-to information is used to know all possible locations from which the calls originate (named o_1 above) and also the locations where the tasks are sent (named o_2 above). Arrows are drawn for all possible combinations. These arrows capture the parallelism in the execution and allow us to gain precision w.r.t. the serial execution. Intuitively, they allow us to consider the maximal cost of the path that continues the execution and the path that goes over the spawned tasks. Finally, dashed edges E_3 are required for expressing the different orderings of the execution of tasks within each abstract location. Without further knowledge, the exit blocks of methods must be joined with the entry blocks of others tasks that execute at the same location. With the MHP analysis we can avoid some dashed edges in the DFG in the following way: given two methods m, whose initial block is m_1, and p, whose final block is p_2, if we know that m_1 cannot happen in parallel with p_2, then we do not need to add a dashed edge between them. This is because the MHP guarantees that when the execution of p finishes there is no instance of method m in the queue of pending tasks. Thus, we do not consider this path in E_3 of the DFG.

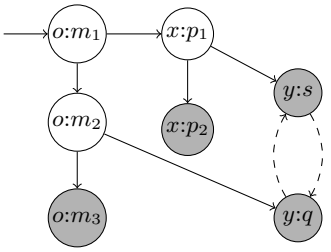


Fig. 3. DFG for Fig. 2

Example 8. Fig. 3 shows the DFG for the program in Fig. 2. The nodes are the cost centers in Ex. 6. Nodes in gray are the nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$, and it implies that the execution can terminate executing $o{:}m_3$, $x{:}p_2$, $y{:}s$ or $y{:}q$. Solid edges include those existing in the CFG of the sequential program but combined with the location's identity (E_1) and those derived from calls (E_2). Since $y{:}s \parallel y{:}q$ (see Ex. 7), the execution order of s and q at location y is unknown (see Sec. 3). This is modelled by means of the dashed edges (E_3). In contrast, since $o{:}m_1 \nparallel o{:}m_3$ and $x{:}p_1 \nparallel x{:}p_2$, we neither add a dashed edge from $o{:}m_3$ to $o{:}m_1$ nor from $x{:}p_2$ to $x{:}p_1$. ∎

5.2 Inference of Parallel Cost

The next phase in our analysis consists of obtaining the maximal parallel cost from all possible executions of the program, based on the DFG. The execution

paths in the DFG start in the initial node that corresponds to the entry method of the program, and finish in any node in $\overline{\mathcal{B}}_{\mathsf{exit}}$. The first step for the inference is to compute the set of execution paths by solving the so-called *single-source path expression problem* [18], which finds a regular expression (named *path expression*) for each node $v \in \overline{\mathcal{B}}_{\mathsf{exit}}$ representing all paths from an initial node to v. Given a DFG $\mathcal{G}$, we denote by $\mathit{pexpr}(\mathcal{G})$ the set of path expressions obtained from the initial node to all exit nodes in $\mathcal{G}$.

Example 9. To compute the set *pexpr* for the graph in Fig. 3, we compute the path expressions starting from $o{:}m_1$ and finishing in exit nodes, that is, the nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$. In path expressions, we use $o{:}m_1{\cdot}o{:}m_2$ to represent the edge from $o{:}m_1$ to $o{:}m_2$. Thus, for the nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$ we have $\mathsf{e}_{o:m_3} = o{:}m_1{\cdot}o{:}m_2{\cdot}o{:}m_3$, $\mathsf{e}_{x:p_2} = o{:}m_1{\cdot}x{:}p_1{\cdot}x{:}p_2$, $\mathsf{e}_{y:s} = o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s \mid o{:}m_2{\cdot}y{:}q{\cdot}y{:}s){\cdot}(y{:}q{\cdot}y{:}s)^*$ and $\mathsf{e}_{y:q} = o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s{\cdot}y{:}q \mid o{:}m_2{\cdot}y{:}q){\cdot}(y{:}s{\cdot}y{:}q)^*$. ∎

The key idea to obtain the parallel cost from path expressions is that the cost of each block (obtained by using the block-level cost analysis) contains not only the cost of the block itself but this cost is multiplied by the number of times the block is visited. Thus, we use sets instead of sequences since the multiplicity of the elements is already taken into account in the cost of the blocks. Given a path expression e, we define $\mathit{sequences}(\mathsf{e})$ as the set of paths produced by e and $\mathit{elements}(p)$ as the set of nodes in a given path p. We use the notions of *sequences* and *elements* to define the set $\mathcal{N}(\mathsf{e})$.

Definition 3. *Given a path expression* e, $\mathcal{N}(\mathsf{e})$ *is the following set of sets:*

$$\{s \mid p \in \mathit{sequences}(\mathsf{e}) \ \wedge \ s = \mathit{elements}(p)\}.$$

In practice, this set $\mathcal{N}(\mathsf{e})$ can be generated by splitting the disjunctions in e into different elements in the usual way, and adding the nodes within the repeatable subexpressions once. Thus, to obtain the parallel cost, it is sufficient to compute $\mathcal{N}^+(\mathsf{e})$, the set of *maximal* elements of $\mathcal{N}(\mathsf{e})$ with respect to set inclusion, i.e., those sets in $\mathcal{N}(\mathsf{e})$ which are not contained in any other set in $\mathcal{N}(\mathsf{e})$. Given a graph $\mathcal{G}$, we denote by $\mathit{paths}(\mathcal{G}) = \bigcup \mathcal{N}^+(\mathsf{e}),\ \mathsf{e} \in \mathit{pexpr}(\mathcal{G})$, i.e., the union of the sets of sets of elements obtained from each path expression.

Example 10. Given the path expressions in Ex. 9, we have the following sets:

$$\mathcal{N}^+(\mathsf{e}_{o:m_3}) = \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1}\}, \quad \mathcal{N}^+(\mathsf{e}_{x:p_2}) = \{\underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2}\}$$

$$\mathcal{N}^+(\mathsf{e}_{y:s}) = \mathcal{N}^+(\mathsf{e}_{y:q}) = \{\underbrace{\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\}}_{N_3}, \underbrace{\{o{:}m_1, o{:}m_2, y{:}s, y{:}q\}}_{N_4}\}$$

Observe that these sets represent traces of the program. The execution captured by N_1 corresponds to trace ① of Fig. 2. In this trace, the code executed at location o leads to the maximal cost. Similarly, the set N_3 corresponds to trace ② and N_4 corresponds to trace ③. The set N_2 corresponds to a trace where $x{:}p_2$ leads to the maximal cost (not shown in Fig. 2). Therefore, the set *paths* is $\{N_1, N_2, N_3, N_4\}$. ∎

Given a set $N \in paths(\mathcal{G})$, we can compute the cost associated to N by using the block-level cost analysis, that is, $\mathcal{S}(\bar{x})|_N$. The parallel cost of the distributed system can be over-approximated by the maximum cost for the paths in $paths(\mathcal{G})$.

Definition 4 (Inferred parallel cost). *The* inferred parallel cost *of a program $P(\overline{x})$ with distributed flow graph $\mathcal{G}$, is defined as* $\widehat{\mathcal{P}}(P(\bar{x})) = \max_{N \in paths(\mathcal{G})} \mathcal{S}(\bar{x})|_N$.

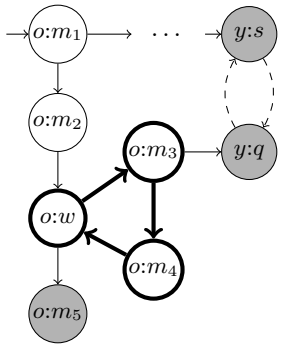
Although we have obtained the parallel cost of the whole program, we can easily obtain the parallel cost associated to a location o of interest, denoted $\widehat{\mathcal{P}}(P(\bar{x}))|_o$, by considering only the paths that lead to the exit nodes of this location. In particular, given a location o, we consider the set of path expressions $pexpr(\mathcal{G}, o)$ which are the subset of $pexpr(\mathcal{G})$ that end in an exit node of o. The above definition simply uses $pexpr(\mathcal{G}, o)$ instead of $pexpr(\mathcal{G})$ in order to obtain $\widehat{\mathcal{P}}(P(\bar{x}))|_o$.

Example 11. The cost is obtained by using the block-level costs for all nodes that compose the sets in *paths*. With the sets computed in Ex. 10, the overall parallel cost is: $\widehat{\mathcal{P}}(\mathsf{m}(n)) = max(\mathcal{S}(n)|_{N_1}, \mathcal{S}(n)|_{N_2}, \mathcal{S}(n)|_{N_3}, \mathcal{S}(n)|_{N_4})$. Importantly, $\widehat{\mathcal{P}}$ is more precise than the serial cost because all paths have at least one missing node. For instance, N_1 does not contain the cost of $x{:}p_1$, $x{:}p_2$, $y{:}s$, $y{:}q$ and N_3 does not contain the cost of $o{:}m_2$, $o{:}m_3$, $x{:}p_2$. Additionally, as $o{:}m_3$ is the only final node for location o, we have that $\widehat{\mathcal{P}}(\mathsf{m}(n))|_o = \mathcal{S}(n)|_{N_1}$. Similarly, for location y we have two exit nodes, $y{:}s$ and $y{:}q$, thus $\widehat{\mathcal{P}}(\mathsf{m}(n))|_y = max(\mathcal{S}(n)|_{N_3}, \mathcal{S}(n)|_{N_4})$. ■

Recall that when there are several calls to a block $o{:}b$ the graph contains only one node $o{:}b$ but the serial cost $\mathcal{S}(\bar{x})|_{o:b}$ accumulates the cost of all calls. This is also the case for loops or recursion. The nodes within an iterative construct form a cycle in the DFG and by setting to one the corresponding cost center, the serial cost accumulates the cost of all executions of such nodes.

Example 12. The program to the right shows a modification of method m that adds a loop which includes the call y.q(). The DFG for this code contains a cycle caused by the loop, composed by the nodes $o{:}w$, $o{:}m_3$ and $o{:}m_4$, where $o{:}w$ represents the entry block to the while loop. The execution might traverse such nodes multiple times and consequently multiple instances of $y{:}q$ might be spawned.

```
void m (int n) {
  ... // m1 instr
  x.p();
  ... //m2 instr
  while(n > 0) {
    n=n−1;
    ... //m3 instr
    y.q();
    ... //m4 instr
  }
  ... // m5 instr
}
```



A serial cost analyzer (e.g.[2]) infers that the loop is traversed at most n times and obtains a block-level serial cost of the form:

$$\mathcal{S}(n) = c(o{:}m_1)*\widehat{m_1} + c(o{:}m_2)*\widehat{m_2} + n*c(o{:}w)*\widehat{w} + n*c(o{:}m_3)*\widehat{m_3} + n*c(o{:}m_4)*\widehat{m_4} + c(o{:}m_5)*\widehat{m_5} + c(x{:}p_1)*\widehat{p_1} + c(x{:}p_2)*\widehat{p_2} + n*c(y{:}q)*\widehat{q} + c(y{:}s)*\widehat{s}$$

For the DFG we obtain some interesting sets that traverse the loop: $N_1 = \{o{:}m_1, o{:}m_2, o{:}w, o{:}m_3, o{:}m_4, o{:}m_5\}$ and $N_2 = \{o{:}m_1, o{:}m_2, o{:}w, o{:}m_3, o{:}m_4, y{:}q, y{:}s\}$. Observe that N_1 represents a trace that traverses the loop and finishes in $o{:}m_5$ and N_2 represents a trace that reaches $y{:}q$ by traversing the loop. The cost associated to N_1 is computed as $\mathcal{S}(n)|_{N_1} = \widehat{m_1}+\widehat{m_2}+n*\widehat{w}+n*\widehat{m_3}+n*\widehat{m_4}+\widehat{m_5}$. Note that $\mathcal{S}(n)|_{N_1}$ includes the cost of executing the nodes of the loop multiplied by n, capturing the iterations of the loop. Similarly, for N_2 we have $\mathcal{S}(n)|_{N_2}=\widehat{m_1}+\widehat{m_2}+n*\widehat{w}+n*\widehat{m_3}+n*\widehat{m_4}+n*\widehat{q}+\widehat{s}$, which captures that q might be executed n times. ∎

Theorem 1. $\mathcal{P}(P(\bar{x})) \leq \widehat{\mathcal{P}}(P(\bar{x}))$.

6 Parallel Cost Analysis with Cooperative Concurrency

We now extend the language to allow cooperative concurrency between the tasks at each location, in the style of concurrent (or active) object systems such as ABS [11]. The language is extended with *future variables* which are used to check if the execution of an asynchronous task has finished. In particular, an asynchronous call is associated with a future variable f as follows f=x.p(). The instruction await f? allows synchronizing the execution of the current task with the task p to which the future variable f is pointing; f.get is used to retrieve the value returned by the completed task. The semantics for these instructions is given in Fig. 4. The semantics of Async+Fut differs from Async in Fig. 1 in that it stores the association of the future variable to the task in the local variable table l. In Await1, the future variable we are awaiting points to a finished task and await can be completed. The finished task t_1 is looked up at all locations in the current state (denoted by Locs). Otherwise, Await2 yields the lock so any other task at the same location can take it. In Get1 the return value is retrieved after the task has finished and in Get2 the location is blocked allowing time to pass until the task finishes and the return value can be retrieved.

Handling concurrency in the analysis is challenging because we need to model the fact that we can lose the processor at the await instructions and another pending task can interleave its execution with the current task. The first extension needed is to refine the block partitioning in Sec. 4 with the set of blocks: $\mathcal{B}_{\mathsf{get}}$, the set of blocks starting with a get; and $\mathcal{B}_{\mathsf{await}}$, the set of blocks starting with an await. Such blocks contain edges to the preceding and subsequent blocks as in the standard construction of the CFG (and we assume they are in the set of edges E_1 of Def. 2). Fortunately, task interleavings can be captured in the graph in a clean way by treating await blocks as initial blocks, and their predecessors as ending blocks. Let b be a block which contains a f.get or await f? instruction. Then $awaited(\mathsf{f}, pp)$ returns the (set of) exit blocks to which the future variable f can be linked at program point pp. We use the points-to analysis results to find the tasks a future variable is pointing to. Furthermore, the MHP analysis learns information from the await instructions, since after an await f? we know that the execution of the task to which f is linked is finished and thus it will not happen in parallel with the next tasks spawned at the same location.

$$\frac{l(x) = lid_1,\ \mathrm{fresh}(tid_1),\ l' = l[f \to tid_1],\ l_1 = buildLocals(\bar{z}, m_1)}{loc(lid, tid, \{tsk(tid, m, l, \langle f = x.m_1(\bar{z}); s\rangle\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1))\})}\ (\textsc{async+fut})$$

$$\frac{l(f) = tid_1,\ loc(lid_1, \_, \mathcal{Q}_1) \in \mathtt{Locs},\ tsk(tid_1, \_, \_, s_1) \in \mathcal{Q}_1,\ s_1 = \epsilon(v)}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, s)\} \cup \mathcal{Q})}\ (\textsc{await1})$$

$$\frac{l(f) = tid_1,\ loc(lid_1, \_, \mathcal{Q}_1) \in \mathtt{Locs},\ tsk(tid_1, \_, \_, s_1) \in \mathcal{Q}_1,\ s_1 \neq \epsilon(v)}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q})}\ (\textsc{await2})$$

$$\frac{l(f) = tid_1,\ tsk(tid_1, \_, \_, s_1) \in \mathtt{Locs}, s_1 = \epsilon(v),\ l' = l[x \to v])}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{x}{=}f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q})}\ (\textsc{get1})$$

$$(\textsc{get2})\ \frac{l(f) = tid_1,\ tsk(tid_1, \_, \_, s_1) \in \mathtt{Locs}, s_1 \neq \epsilon(v)}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{x}{=}f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{x}{=}f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q})}$$

Fig. 4. Summarized Semantics of Concurrent Execution

Definition 5 (DFG with cooperative concurrency). *We extend Def. 2:*
$E_4 = \{o_1{:}m_{exit} \to o_2{:}b_2 \mid \textit{either } pp{:}f.\mathsf{get} \textit{ or } pp{:}\mathsf{await}\ f? \in b_2, m_{exit} \in awaited(f, pp)\}$
$E_5 = \{o{:}b_1 \dashrightarrow o{:}b_2 \mid b_1 \in pred(\mathcal{B}_{await}), b_2 \in \mathcal{B}_{await} \cup \mathcal{B}_{init}, o{:}b_1 \parallel o{:}b_2\}$
$E_6 = \{o{:}b_1 \dashrightarrow o{:}b_2 \mid b_1 \in \mathcal{B}_{exit}, b_2 \in \mathcal{B}_{await}, o{:}b_1 \parallel o{:}b_2\}$

Here, E_4 contains the edges that relate the last block of a method with the corresponding synchronization instruction in the caller method, indicating that the execution can take this path after the method has completed. E_5 and E_6 contain dashed edges that represent the orderings between parts of tasks split by await instructions and thus capture the possible interleavings. E_5 considers the predecessor as an ending block from which we can start to execute another interleaved task (including await blocks). E_6 treats await blocks as initial blocks which can start their execution after another task at the same location finishes. As before, the MHP analysis allows us to discard those edges between blocks that cannot be pending to execute when the processor is released. Theorem 1 also holds for DFG with cooperative concurrency.

Example 13. Fig. 5 shows an example where the call to method p is synchronized by using either await or get. Method p then calls method q at location o. The synchronization creates a new edge (the thick one) from $x{:}p_2$ to the synchronization point in block $o{:}m_3$. This edge adds a new path to reach $o{:}m_3$ that represents a trace in which the execution of m waits until p is finished. For the graph in Fig. 5 we have that *paths* is $\{\{o{:}m_1, x{:}p_1, x{:}p_2, o{:}m_3, o{:}q\}, \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}q\}\}$. Observe that the thick edge is crucial for creating the first set in *paths*. The difference between the use of await and get is visible in the edges labelled with ⊛, which are only added for await. They capture the traces in which the execution of m waits for the termination of p, and q starts its execution interleaved between $o{:}m_2$ and $o{:}m_3$, postponing the execution of $o{:}m_3$. In this example, the edges labelled with ⊛ do not produce new sets in *paths*. ∎

Finally, let us remark that our work is parametric in the underlying points-to and cost analyses for serial execution. Hence, any accuracy improvement in these auxiliary analyses will have an impact on the accuracy of our analysis. In particular, a context-sensitive points-to analysis [15] can lead to big accuracy gains. Context-sensitive points-to analyses use the program point from which tasks are spawned as context information. This means that two different calls $o.m$, one from program point p_1 and another from p_2 (where $p_1 \neq p_2$) are distinguished in the analysis as $o{:}p_1{:}m$ and $o{:}p_2{:}m$. Therefore, instead of representing them by a single node in the graph, we will use two nodes. The advantage of this finer-grained information is that we can be more accurate when considering task parallelism. For instance, we can have one path in the graph which includes a single execution of $o{:}p_1{:}m$ (and none of $o{:}p_2{:}m$). However, if the nodes are merged into a single one, we have to consider either that both or none are executed. There are also techniques to gain precision in points-to analysis in the presence of loops [16] that could improve the precision of our analysis.

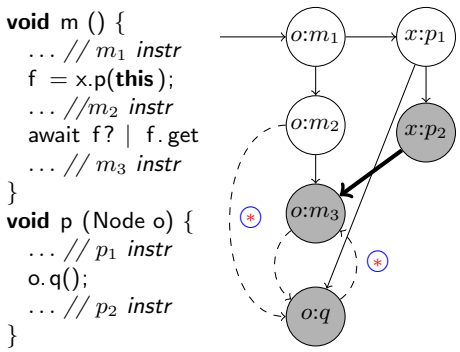


Fig. 5. DCG with synchronization

7 Experimental evaluation

We have implemented our analysis in SACO and applied it to some distributed based systems: BBuffer, the typical bounded-buffer for communicating several producers and consumers; MailServer, which models a distributed mail server system with multiple clients; Chat, which models chat application; DistHT, which implements and uses a distributed hash table; BookShop, which models a web shop client-server application; and P2P, which represents a peer-to-peer network formed by a set of interconnected peers. Experiments have been performed on an Intel Core i7 at 2.0GHz with 8GB of RAM, running Ubuntu 14.04. Table 1 summarizes the results obtained for the benchmarks. Columns Benchmark and loc show, resp., the name and the number of program lines of the benchmark. Columns $\#_{\mathsf{N}}$ and $\#_{\mathsf{E}}$ show the number of nodes and edges of the DFG with concurrency (Def. 5). Columns $\#_{\mathsf{F}}$ and $\#_{\mathsf{P}}$ contain the number of terminal nodes in the DFG and the number of elements in the set *paths*. Columns $\mathbf{T}_{\mathcal{S}}$ and $\mathbf{T}_{\widehat{\mathcal{P}}}$ show, resp., the analysis times for the serial cost analysis and the additional time required by the parallel cost analysis (in milliseconds) to build the DFG graphs and obtain the cost from them. The latter includes a simplification of the DFG to reduce the strongly connected components (SCC) to one node. Such simplification significantly reduces the time in computing the path expressions and we can see that the overall overhead is reasonable.

Column $\%_{\widehat{\mathcal{P}}}$ aims at showing the gain of the parallel cost $\widehat{\mathcal{P}}$ w.r.t. the serial cost $\mathcal{S}$ by evaluating $\widehat{\mathcal{P}}(\bar{e})/\mathcal{S}(\bar{e})*100$ for different values of $\bar{e}$. Namely, $\%_{\widehat{\mathcal{P}}}$ is the av-

Benchmark	loc	$\#_{\mathsf{N}}$	$\#_{\mathsf{E}}$	$\#_{\mathsf{F}}$	$\#_{\mathsf{P}}$	$\mathbf{T}_{\mathcal{S}}$	$\mathbf{T}_{\widehat{\mathcal{P}}}$	$\#_{\mathsf{I}}$	$\%_{\mathsf{m}}$	$\%_{\mathsf{a}}$	$\%_{\widehat{\mathcal{P}}}$
BBuffer	105	37	50	7	50	256	26	1000	3.0	19.7	77.4
MailServer	115	28	35	6	36	846	12	1000	61.1	68.6	88.5
Chat	302	84	245	25	476	592	126	625	5.7	56.0	85.4
DistHT	353	38	47	6	124	950	49	625	3.7	25.5	76.3
BookShop	353	60	63	7	68	2183	214	2025	9.2	50.9	71.1
P2P	240	168	533	27	730	84058	1181	512	13.0	85.9	95.2

Table 1. Experimental results (times in ms)

erage of the evaluation of the cost expressions $\widehat{\mathcal{P}}(\bar{e})$ and $\mathcal{S}(\bar{e})$ for different values of the input arguments $\bar{e}$ to the programs. The number of evaluations performed is shown in column $\#_{\mathsf{I}}$. The accuracy gains range from 4.8% in P2P to 28.9% in BookShop. The gain of more than 20% for DistHT, BookShop and BBuffer is explained by the fact that these examples take advantage of parallelism: the different distributed locations execute a similar number of instructions and besides their code mostly runs in parallel. MailServer, Chat and P2P achieve smaller gains because the blocks that are not included in the path (those that are guaranteed to happen in parallel with longer blocks) are non-recursive. Thus, when the number of instructions is increased, the improvements are reduced proportionally. Moreover, Chat and P2P create very dense graphs, and the paths that lead to the maximum cost include almost all nodes of the graph. Column $\%_{\mathsf{m}}$ shows the ratio obtained for the location that achieves the maximal gain w.r.t. the serial cost. In most examples, except in MailServer, such maximal gain is achieved in the location that executes the entry method. MailServer uses synchronization in the entry method that leads to a smaller gain. Column $\%_{\mathsf{a}}$ shows the average of the gains achieved for all locations. The average gain ranges from 80.3% to 31.4%, except for P2P, which has a smaller gain 14.1% due to the density of its graph as mentioned above.

8 Conclusions and Related Work

We have presented what is to the best of our knowledge the first static cost analysis for distributed systems which exploits the parallelism among distributed locations in order to infer a more precise estimation of the parallel cost. Our experimental results show that parallel cost analysis can be of great use to know if an application succeeds in exploiting the parallelism of the distributed locations. There is recent work on cost analysis for distributed systems which infers the peak of the *serial* cost [3], i.e., the maximal amount of resources that a distributed component might need along its execution. This notion is different to the parallel cost that we infer since it is still serial; i.e., it accumulates the resource consumption in each component and does not exploit the overall parallelism as we do. Thus, the techniques used to obtain it are also different: the peak cost is obtained by abstracting the information in the queues of the different locations using graphs and finding the cliques in such graphs [3]. The only common part with our analysis is that both rely on an underlying resource analysis for the serial execution that uses cost centers and on a MHP analysis,

but the methods used to infer each notion of cost are fundamentally different. This work is improved in [4] to infer the peak for non-cumulative resources that increase and decrease along the execution (e.g., memory usage in the presence of garbage collection). In this sense, the notion of parallel cost makes sense only for cumulative resources since its whole purpose is to observe the efficiency gained by parallelizing the program in terms of resources used (and accumulated) in parallel by distributed components. Recent work has applied type-based amortized analysis for deriving bounds of parallel first-order functional programs [10]. This work differs from our approach in the concurrent programming model, as they do not allow explicit references to locations in the programs and there is no distinction between blocking and non-blocking synchronization. The cost measure is also quite different from the one used in our approach.

To simplify the presentation, we have assumed that the different locations execute one instruction in one cost unit. This is without loss of generality because if they execute at a different speed we can weight their block-level costs according to their relative speeds. We argue that our work is of wide applicability as it can be used in combination with any cost analysis for serial execution which provides us with cost information at the level of the required fragments of code (e.g., [8,9,21]). It can also be directly adopted to infer the cost of parallel programs which spawn several tasks to different processors and then use a join operator to synchronize with the termination of all of them (the latter would be simulated in our case by using a `get` instruction on all spawned tasks). As future work, we plan to incorporate in the analysis information about the scheduling policy used by the locations (observe that each location could use a different scheduler). In particular, we aim at inferring (partial) orderings among the tasks of each location by means of static analysis.

Analysis and verification techniques for concurrent programs seek finite representations of the program traces to avoid an exponential explosion in the number of traces (see [7] and its references). In this sense, our DFG's provide a finite representation of all traces that may arise in the distributed system. A multithread concurrency model entails an exponential explosion in the number of traces, because task scheduling is preemptive. In contrast, cooperative concurrency as studied in this paper limits is gaining attention both for distributed [11] and for multicore systems [6,20], because the amount of interleaving between tasks that must be considered in analyses is restricted to synchronization points which are explicit in the program.

References

1. WCET tools. http://www.rapitasystems.com/WCET-Tools, 2012.
2. E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.
3. E. Albert, J. Correas, and G. Román-Díez. Peak Cost Analysis of Distributed Systems. In *Proc. of SAS'14*, volume 8723 of *LNCS*, pages 18–33, 2014.
4. E. Albert, J. Correas, and G. Román-Díez. Non-Cumulative Resource Analysis. In *Procs. of TACAS'15*, volume 9035 of *LNCS*, pages 85–100. Springer, 2015.

5. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
6. S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. Broch Johnsen, Ka I Pun, S. Lizeth Tapia Tarifa, T. Wrigstad, and A. Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 1–56. Springer, 2015.
7. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142. ACM, 2013.
8. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
9. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.
10. J. Hoffmann and Z. Shao. Automatic Static Cost Analysis for Parallel Programs. In *Procs. of ESOP'15*, volume 9032 of *LNCS*, pages 132–157. Springer, 2015.
11. E. Broch Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Procs. of FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
12. J. K. Lee, J. Palsberg, and R. Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.
13. 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.
14. M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997. ACM.
15. 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.
16. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
17. Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
18. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.
19. B. Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.
20. J. Yi, C. Sadowski, S. N. Freund, and C. Flanagan. Cooperative concurrency for a multicore world - (extended abstract). In *Procs. of RV'11*, volume 7186 of *LNCS*, pages 342–344. Springer, 2012.
21. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

Appendix I

Article *Resource consumption of concurrent objects over time*, [19]

Resource Consumption of Concurrent Objects over Time

Antonio Flores-Montoya and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science
aeflores|haehnle@cs.tu-darmstadt.de

Abstract. We present the first static resource analysis of timed concurrent object programs. Instead of measuring the total resource consumption over a complete execution, we measure the resource consumption at different moments in time, that is, the *resource consumption over time*. To obtain such a measure we perform a program transformation that generates a program without time whose resource consumption corresponds to the resource consumption of the timed program during time t. The transformed program can then be analyzed with a combination of existing tools. These provide upper bounds that safely approximate the resource consumption of all possible behaviors of the program at all possible times. We implemented a prototype of the approach and evaluated it on a complex program to demonstrate its feasibility.

1 Introduction

The use of static analysis to infer upper bounds on the resource consumption of software systems is a very active area of research. Many tools have been recently developed [7, 8, 10, 11, 15]. However, most effort has been focused on the analysis of sequential programs. There is some initial research that tries to apply resource analysis techniques to distributed and concurrent systems [1, 2, 12]. In contrast to sequential systems, in a concurrent system it is not only important to estimate the amount of consumed resources but also *when* they are consumed. If we have a set of tasks to be executed, a system will behave differently, depending on whether the tasks are scheduled simultaneously, sequentially, etc. Some aspects of this richer concept of resource consumption are already explored in [2, 12].

We propose a novel approach for measuring concurrent resource consumption based on a discrete time model. We present a static analysis that infers upper bounds of the *resource consumption over time*. These upper bounds are expressed as a function of the entry parameters of the program and the elapsed time and allow to explore how resource consumption varies in a given program as time advances. In our program the explicit aspects of time are captured in two primitives, **await duration**(t) and **until**(t'), that suspend a task for a certain period of time t or until the absolute time has reached t'. The main advantage of this approach is that, by simply placing these time primitives at different locations inside a program, we can obtain different kinds of behavior and analyze their resource consumption.

The target of our analysis is a language based on concurrent objects. In this language, each object owns a processor and executes in parallel with others. Each object contains a set of tasks and only one task per object can be executed at any given time, while the scheduling of tasks is non-deterministic and non-preemptive. Communication and synchronization among objects is based on asynchronous message passing and future variables. The language is inspired by ABS [13] and the time primitives are (slightly simplified) taken from [6, 14]. However, the presented approach could be easily adopted to other concurrency models based on creating and joining tasks.

To the best of our knowledge, our paper presents the first static resource analysis that analytically derives sound symbolic upper bounds for timed concurrent programs. The analysis generates a set of upper bounds that summarize analytically the resource consumption of a program over time. The main contributions of this work are:

- We define the novel concept of *resource consumption over time* (Sec. 3).
- We develop a sound program transformation that generates an untimed program from a timed program (Sec. 4).
- We show how we can analyze the resulting untimed program with a combination of existing tools and how the results can be interpreted (Sec. 5).
- We illustrate the flexibility of the timed approach by inferring the *peak cost* of an example borrowed from [2] (Sec. 6).

2 Timed Concurrent Programs

We adopt a lightweight object-oriented language. A program $\mathcal{P}$ consists of a set of classes, each of them defines a set of fields and a set of methods. The set of types includes the class names, the integer (Int) primitive type, the set of *future* variable types $fut(T)$ and $Unit$. The latter is the default return type of methods (like $Void$ in C). The notation $\overline{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. Pure expressions (p) have no side effects. A pure expression is *local* (naming convention p_l) if it does not depend on any field. The abstract syntax of class declarations CL, method declarations M, types T, variables v (x for local variables), and statements s is:

$$
\begin{array}{l}
CL ::= \textsf{class}\ C\ \{\overline{T}\ \overline{f}; \overline{M}\} \quad M ::= T\ m(\overline{T}\ \overline{x})\{s; \textsf{return}\ p\} \quad v ::= x \mid \textsf{this}.f \\
s ::= s; s \mid v = e \mid \textsf{if}\,(p)\ s\ \textsf{else}\ s \mid \textsf{while}(p)\ s \\
\qquad \textsf{await}\ x? \mid \textsf{await duration}(p_l) \mid \textsf{until}\,(p_l) \mid \textsf{release} \mid \textsf{cost}\ p \\
e ::= \textsf{new}\ C(\overline{p}) \mid x!m(\overline{p}) \mid x.\textsf{get} \mid p \quad T ::= C \mid Int \mid fut(T) \mid Unit
\end{array}
$$

We assume that all methods have a single return instruction at the end of the method. If the method returns $Unit$, the return instruction is empty *return*. Each object encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the **this** object, and any other object can only access such fields through method calls. We assume that each program includes a method called $\mathsf{main}(\overline{z})$ with a set of initial parameters $\overline{z}$, from which

```
class Hndset(ts,smss){
Unit normalBehavior(mx,dur,tc){
  if (now()>tc && now()<tc+20)
    midnightWindow(mx,dur,tc);
  else{
    ts!call(dur);
    await duration(1);
    normalBehavior(mx,dur,tc);
 }}
Unit midnightWindow(mx,dur,tc){
   if (now() >= tc+20) {
    normalBehavior(mx,dur,tc);
   } else {
     Int i = 0;
     while (i < mx) {
       smss!sendSMS();
       i = i + 1;
     }
    await duration(1);
    midnightWindow(mx,dur,tc);
 }}
}
```

```
class PhoneSvr{
  Unit call(dur){
    while (dur>0) {
     cost 1;
     dur = dur-1;
     await duration(1);
  }}
}

class SMSSvr{
  Unit sendSMS() {
    cost 1;
  }
}

Unit main(mx,dur,tc){
 SMSSvr sms=new SMSSvr();
 PhoneSvr ts=new PhoneSvr();
 Hndset hs=new Hndset(ts,sms);
 hs!normalBehavior(mx,dur,tc);
}
```

Fig. 1. A timed program

execution will start. The main method does not belong to any class and has no fields. The concurrency model is as follows: each object has a lock that is shared by all the tasks that belong to the object. Data synchronization is by means of future variables (denoted y): an **await** y? instruction is used to synchronize with the result of executing a task $y = x!m(\overline{p})$, where **await** y? is suspended until the result assigned to the future variable y is available (i.e., the task is finished). During suspension the object's lock is released so that another pending task on that object can take it. In contrast to await, the expression y.**get** blocks its object (no other task of the same object can run) until y is available. Finally, the instruction **release** releases the object's lock unconditionally.

Time is a discrete magnitude. A timed program has a global clock common to all concurrent components. The current time can be accessed through the pure expression $now()$ (which simply reads the value of the clock). Time is advanced through the primitive **await duration**(p_l) which releases the object's lock until time has advanced p_l units (relative time progress) or **until**(p_l) which releases the object's lock until time is at least p_l (absolute time progress). Time only advances when no task can progress any further.

2.1 Explicit Cost Model

We adopt the standard approach to obtain a parametric cost model. We introduce a new statement **cost** p where p is a pure expression of integer type. When

a statement **cost** p is executed, the result of evaluating p determines the amount of resources consumed. The choice of the cost model (where the cost statements are introduced) determines the resources that we want to observe. For example, if we introduce a statement **cost** 1 after every instruction, we will infer an upper bound on the number of executed instructions. Or we could add a cost instruction to each **new** C instruction to measure the number of objects created. Both time and cost annotations can be introduced automatically according to the underlying architecture, network topology, or any other given criterion.

Example 1. Fig. 1 illustrates an example of a timed program with cost annotations. The return instructions have been omitted given that all the methods return $Unit$. The parameter types have also been omitted (they are all integers). The program contains 3 classes: A phone server PhoneSvr that might process calls of different duration. Each call lasts calltime time units and consumes one resource per time unit; An SMS server SMSSvr that can process SMSs. Each SMS is processed instantly consuming a resource unit. And the class Hndset that models a possible behavior of the clients. In particular, it simulates a scenario where the servers are receiving calls. The duration of calls is given by the parameter dur. At time tc, the behavior changes and we enter the midnight window where we receive mx SMS per time unit. This exceptional behavior lasts until time is $tc+20$ when it changes back to normal. This behavior is modelled by two mutually recursive methods normalBehavior and midnightWindow. Finally, the main method creates an SMS server sms, a phone server ts and calls normalBehavior.

2.2 Operational Semantics

A *program state* S is a set $S = \texttt{Ob} \cup \texttt{T} \cup \{clk(t)\}$ where Ob is the set of all created objects, T is the set of tasks (including finished tasks) and $clk(t)$ is the global clock with the current time t. The associative and commutative union operator on states is denoted by white-space. An *object* is a term $ob(o, a, lk)$ where o is the object identifier, a is a mapping from the object fields to their values, and lk the identifier of the *active task* that holds the object's lock or $\bot$ if the object's lock is free. Only one task can be *active* (running) in each object and hold its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $tsk(tk, o, l, s)$ where tk is a unique task identifier, o identifies the object to which the task belongs, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(val)$ if the task has terminated and the return value val is available. Created objects and tasks never disappear from the state in the semantics.

Given a program $\mathcal{P}$ with a main method $\mathsf{main}(\overline{z})$ and a set of initial values $\overline{val}$, the execution of a program starts from the initial state $S_0(\overline{val}) = \{obj(0, f, \bot), tsk(0,\ 0, buildLoc(\overline{val}, main), body(\mathsf{main})), clk(1)\}$ where we have an initial object with identifier 0 with a free lock $\bot$. f is an empty mapping (since main has no fields). The local state is generated by $buildLoc(\overline{val}, \mathsf{main})$ that maps the main method parameters $\overline{z}$ to the given entry values $\overline{val}$ and the rest

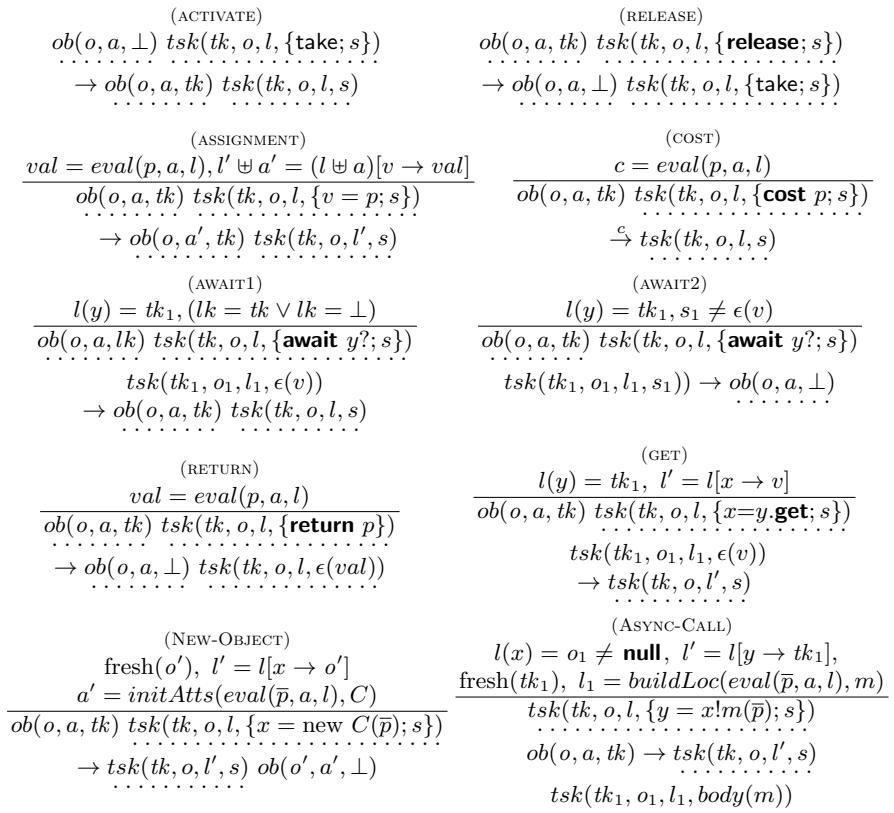


Fig. 2. Semantics

of local variables to default values. The initial time is one. Given a method $M ::= T\ m(\overline{T}\ \overline{x})\{s; \textbf{return}\ p\}$, the function $body(M)$ returns $\{\mathsf{take}; s; \textbf{return}\ p\}$ i.e. the sequence of statements of the method preceded by an auxiliary instruction take that fetches the lock.

Execution proceeds from S_0 by applying *non-deterministically* the semantic rules depicted in Figs. 2 and 3. We omit the rules for **if** and **while** as they are standard, they can be found in the App. B. The operational semantics is given in a rewriting style where a step is a transition of the form $a\underline{b} \xrightarrow{c} \underline{b'}\ n$, where the dotted underlining indicates that term b is rewritten into b′; we look up the term a but do not modify it and hence it is not included in the subsequent state; term n is newly added to the state; and c is the cost of the transition according to the cost model. We omit c if it is zero. For simplicity, we assume that only local variables are assigned in the rules New-Object,Async-Call and Get. Fields can still be modified with the rule Assignment. Transitions are applied according to the rules as follows.

$$\frac{t' = t}{tsk(tk, o, l, \{\textbf{await duration}(p_l); s\}) \; ob(o, a, tk) \; clk(t) \rightarrow tsk(tk, o, l, \{\textbf{until}\,(p_l + t'); s\})} \; (\text{AWAIT DURATION})$$

$$\frac{t' = eval(p_l, l), \; t' > t}{tsk(tk, o, l, \{\textbf{until}\,(p_l); s\}) \; ob(o, a, tk) \; clk(t) \rightarrow ob(o, a, \bot) \; tsk(tk, o, l, \{\textsf{untilp}\,(p_l); s\})} \; (\text{UNTIL2})$$

$$\frac{t' = eval(p_l, l), \; t' \leq t}{ob(o, a, tk) \; tsk(tk, o, l, \{\textbf{until}\,(p_l); s\}) \; clk(t) \rightarrow tsk(tk, o, l, s)} \; (\text{UNTIL1})$$

$$\frac{t' = eval(p_l, l), \; t' \leq t}{ob(o, a, \bot) \; tsk(tk, o, l, \{\textsf{untilp}\,(p_l); s\}) \; clk(t) \rightarrow ob(o, a, tk) \; tsk(tk, o, l, s)} \; (\text{UNTIL3})$$

$$\frac{canAdv, t' = t + 1}{clk(t) \rightarrow clk(t')} \; (\text{TICK})$$

Fig. 3. Time semantics

ACTIVATE: A task that has a take statement obtains its object's lock if it is free.
RELEASE: **release** unconditionally yields the object's lock so any other task of the same object can take it. The instruction take is added to the pending instructions.
ASSIGNMENT: The variable v gets assigned to the result of evaluating the pure expression p given the current state a and l (denoted $eval(p, a, l)$). The notation $l' = l[v \rightarrow val]$ denotes that the mapping l' is equal to l for all values except v where $l'(v) = val$. We assume local variables and fields are always different. If v is a local variable, l is updated $l' = l[v \rightarrow val]$. If v is a field, a is updated $a' = a[v \rightarrow val]$. We express that as $l' \uplus a' = (l \uplus a)[v \rightarrow val]$.
COST: c resource units are consumed where c is the result of evaluating the expression p in the current state. As the object o is not modified, it is not included in the resulting state.
AWAIT1: The (local) future variable we are waiting for points to a finished task and the await is completed. The finished task t_1 is only looked up but it does not disappear from the state as its return value may be needed later on. To complete the await, the object's lock must be free or in the task tk. As a result of applying the rule, the task tk obtains (or keeps) the object's lock.
AWAIT2: If the task we are awaiting for is not finished, We release the lock.
RETURN: When **return** is executed, the return value val is stored (by adding the instruction $\epsilon(val)$) so that it can be obtained by the future variables that reference the task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is finished.
GET: An $x = y.\textbf{get}$ instruction waits for the (local) future variable but without yielding the lock. It stores the value associated with the future variable y in x.
NEW-OBJECT: An active task tk in object o creates an object o' of type C, its fields are initialized with default values and the given parameters $eval(\overline{p}, a, l)$ ($initAtts$) and o' is introduced to the state with a free lock.

ASYNC_CALL: A method call creates a new task (the initial state is created by *buildLoc* using the calling values $eval(\overline{p}, a, l)$) with a fresh task identifier t_1 which is associated to the corresponding future variable y in l'.
AWAIT DURATION: It generates an $\textbf{until}(p_l + t')$ instruction that substitutes the original $\textbf{await duration}(p_l)$ where t' is a constant with the current time of the clock.
UNTIL1: If the current time is greater or equal than the time we are waiting for and the task has the object's lock, we complete the $\textbf{until}(p_l)$ instruction. The time we are waiting for is the result of evaluating the pure local expression p_l. We write $eval(p_l, l)$ to emphasize that p_l does not depend on the object's state.
UNTIL2: If we have not reached the time we are waiting for, we release the lock. we substitute $\mathsf{untilp}(p_l)$ for $\textbf{until}(p_l)$. $\mathsf{untilp}(p_l)$ will be able to take back the lock. Its behavior is analogous to the auxiliary instruction take with respect to **release**.
UNTIL3: If the current time is greater or equal than the time we are waiting for and the lock is free, we complete the instruction $\mathsf{untilp}(p_l)$ and obtain the lock.
TICK: The rule tick makes the time advance one unit. It is only applicable if no task can progress. This behavior reflects the run-to-completion policy and is enforced by the function *canAdv*. The latter is true only when *no other semantic rule can be applied* to any task of the current state.

3 Cost over Time

The traditional definition of cost used in static resource analysis is concerned with the total amount of resources consumed during the complete execution of a program. Given program $\mathcal{P}$ and input values $\overline{val}$, we can define the cost of a trace as follows:

Definition 1 (Cost of a trace). *Let* $tr(\overline{val}) = S_0(\overline{val}) \xrightarrow{c_0} S_1 \xrightarrow{c_1} \cdots$ *be a (possibly infinite) execution trace where* $\overline{val}$ *are the input values of the main method and* c_i *the cost of the semantic transition leading to state* S_{i+1}*. The cost of executing* $tr(\overline{val})$ *is* $Cost_{tr(\overline{val})} = \sum_i c_i$ *i.e. the sum of the costs of the all transitions in the trace* tr*.*

Let $\mathcal{TR}_\mathcal{P}$ be the set of all possible traces of a program for all possible input values, we define an upper bound of such a program:

Definition 2 (Upper bound). *Let* $\overline{T}$ *be the types of the input parameters of a program* $\mathcal{P}$*, an upper bound function is defined as* $ub : \overline{T} \to \mathbb{R}$*.* $ub(\overline{x})$ *is a valid upper bound of* $\mathcal{P}$ *iff for all input values* $\overline{val}$ *and traces* $tr(\overline{val}) \in \mathcal{TR}_\mathcal{P}$*:* $Cost_{tr(\overline{val})} \leq ub(\overline{val})$*.*

A *conditional upper bound* is an upper bound *ub* with a precondition such that the upper bound is only valid when the input parameters satisfy the precondition.

The main problem with this upper bound definition is that it does not give any information on how and *when* the resources are consumed. In the example from Fig. 1 the total resource consumption is not bounded since it consists of an

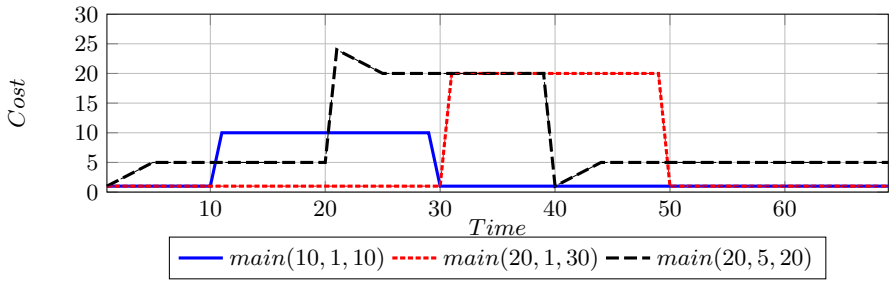


Fig. 4. Resource consumption profiles of the model from Fig. 1 which coincide with their inferred upper bounds

infinite execution that consumes a non-zero amount of resources at each iteration. For programs with timed behavior, we are interested in their consumption over time, that is, the maximum possible amount of resources consumed at a moment in time t.

Example 2. In Fig. 4, we show the resource consumption over time of our program from Fig. 1 given different possible input parameters. We generated values by executing the program with specific input values, and derived concrete values from our *analytically obtained* upper bounds. In all three cases, the upper bounds are precise. The first parameter of main determines the height of the midnight window, the second the duration of the calls, and the third when the midnight window takes place. Note that when a call is started, its cost is distributed in time. In the third profile, we can see how at the start of the midnight window there are still some calls in progress which cause an extra peak in resource consumption. On the other hand, when the midnight window ends, it takes some time until the cost goes back to 5 as calls are started one at a time.

Definition 3 (Cost of a trace in time). *Let* $tr(\overline{val}) = S_0(\overline{val}) \xrightarrow{c_0} S_1 \xrightarrow{c_1} \cdots$ *be an execution trace as above. The cost of executing such a trace at time* tg *is the sum of the cost of all transitions where the time is* tg*.* $Cost_{tr(\overline{val})}(tg) = \sum_{i|clk(tg)\in S_{i+1}} c_i$.

The goal of our analysis is to approximate (safely) the behavior of systems over time by obtaining a set of (conditional) upper bounds that are parametrized by the time we want to observe.

Definition 4 (Time-parametrized upper bound). *Let* $\overline{T}$ *be the types of the input parameters of a program* $\mathcal{P}$*. A time-parametrized upper bound function has type* $ub : (\overline{T}, \mathbb{N}) \to \mathbb{R}$*. Then* $ub(\overline{x}, tg)$ *is a valid time-parametrized upper bound of* $\mathcal{P}$ *iff for all input values* $\overline{val}$*,* $tg \in \mathbb{N}$ *and* $tr(\overline{val}) \in \mathcal{TR}_{\mathcal{P}}$*:* $Cost_{tr(\overline{val})}(tg) \leq ub(\overline{val}, tg)$.

$$
\begin{array}{lll}
1:\alpha(\textbf{cost}\ c) & \Rightarrow \textbf{if}\,(\mathsf{tg}{==}\textbf{now}())\ \textbf{cost}\ \mathsf{c}; & \\
2:\alpha(y = x!m(\overline{v})) & \Rightarrow y = x!m(\overline{v}, tg) & \\
3:\alpha(s_1; s_2) & \Rightarrow \alpha(s_1); \alpha(s_2) & \\
4:\alpha(\textbf{if}\,(p)\ s_1\ \textbf{else}\ s_2) & \Rightarrow \textbf{if}\,(p)\ \alpha(s_1)\ \textbf{else}\ \alpha(s_2) & \\
5:\alpha(\textbf{while}(p)\ s) & \Rightarrow \textbf{while}(p)\ \alpha(s) & \\
6:\alpha(s) & \Rightarrow s & \text{otherwise}
\end{array}
$$

Fig. 5. Transformation over the statements

4 Program Transformation

To analyze timed programs we use a sound program transformation that generates untimed programs which can be analyzed by standard tools. In a first step (Sec. 4.1) we transform a given program $\mathcal{P}$ into a program $\mathcal{P}_{tg}$ such that the resource consumption of $\mathcal{P}$ at time tg is the same as the total resource consumption of $\mathcal{P}_{tg}$. Then we perform a second transformation (Sec. 4.2) that safely models the advance of time.

4.1 Cost Model Specialization

To achieve the desired effect, every method (including the main method) is equipped with an extra parameter tg that holds the time we want to observe. We assume tg is disjoint with the existing variable names. The statements of each method are transformed using the function α defined in Fig. 5. A method declaration $T\ m(\overline{T}\ \overline{x})\{s; \textbf{return}\ p;\}$ becomes $T\ m(\overline{T}\ \overline{x}, Int\ tg)\{\alpha(s); \textbf{return}\ p;\}$.

The function α defined in Fig. 5 works as follows: (1) we substitute every cost statement by a conditional statement that only consumes resources if the current time is tg; (2) tg is added to the method calls as a parameter; (3),(4) and (5) the tranformation of non-atomic statements is the transformation of the statements that compose them; and (6) the remaining statements are not modified.

To prove soundness of the transformation, we define the concept of extended local state and α-equivalent state.

Definition 5 (Extended local state). *Let l be a local state of a task, then $l_{tg} = l + [tg \to y]$ is a mapping where for every $x \in Dom(l)$, $l(x) = l_{tg}(x)$ and $l_{tg}(tg) = y$. We call l_{tg} an extended local state of l with tg.*

Definition 6 (α-equivalent state). *Define an extended execution state S^α:*
$S^\alpha = \{e \mid e \in S \wedge e \neq tsk(tk, o, l, s)\} \cup$
$\quad \{tsk(tk, o, l + [tg \to tg_o], \alpha(s)) \mid tsk(tk, o, l, s) \in S\}$
We apply the transformation α to all the pending statements of all the tasks in S and take their extended local state. The objects and the clock remain unchanged; tg_0 denotes the unique initial value of the variable tg throughout the execution.

We use the function α for two purposes: the function defines the transformation over the original program and, at the same time, we use it to establish a relation between execution states of the original and transformed program.

$$
\begin{array}{lll}
1 : \tau(y = x!m(\overline{p})) & \Rightarrow y = x!m(\overline{p}, time) & \\
2 : \tau(\textsf{until}\,(p_l)) & \Rightarrow release; time = max(p_l, time); & \\
3 : \tau(\textsf{await duration}(p_l)) & \Rightarrow release; time = max(time + p_l, time); & \\
4 : \tau(\textsf{await}\ y?) & \Rightarrow \textsf{await}\ y?; time = max(time, y.time); & \\
5 : \tau(\textsf{until}\mathsf{p}\,(t)) & \Rightarrow take; time = max(t, time); & \\
6 : \tau(s_1; s_2) & \Rightarrow \tau(s_1); \tau(s_2) & \\
7 : \tau(\textsf{if}\,(p)\ s_1\ \textsf{else}\ s_2) & \Rightarrow \textsf{if}\,(p)\ \tau(s_1)\ \textsf{else}\ \tau(s_2) & \\
8 : \tau(\textsf{while}(p)\ s) & \Rightarrow \textsf{while}(p)\ \tau(s) & \\
9 : \tau(s) & \Rightarrow s & \text{otherwise}
\end{array}
$$

Fig. 6. Second transformation over the statements

Theorem 1 (Soundness). *Given a program $\mathcal{P}$ and its transformed program $\mathcal{P}_{tg}$, for every trace $tr(\overline{val}) \in \mathcal{TR}_{\mathcal{P}}$ whose final state is S, there is a trace $tr'(\overline{val}, tg_0) \in \mathcal{TR}_{\mathcal{P}_{tg}}$ whose final state is S^α and $Cost_{tr(\overline{val})}(tg_0) = Cost_{tr'(\overline{val}, tg_0)}$.*

This states that each behavior of the original program can be simulated by the transformed program. In addition, the transformed program $\mathcal{P}_{tg}$ captures the amount of the total cost consumed at time tg_0 (the input parameter).

Proof idea By induction on the length of a trace. We show that for each step $S \xrightarrow{c} S_{new}$ in $\mathcal{P}$, there is a step or a sequence of steps in the transformed program between α-equivalent states $S^\alpha \xrightarrow{c_1} S_1 \xrightarrow{c_2} \cdots \xrightarrow{c_m} S^\alpha_{new}$. Moreover, the cost $\sum_{i=1}^{m} c_i$ is c if $clk(tg_0) \in S$ and 0 if $clk(t') \in S$ for $t' \neq tg_0$. We can apply the same semantic rules in the original and the transformed program for all cases except for rule (Cost) and reach an α-equivalent state (the addition of the variable tg does not affect the behavior). For rule (Cost) we show that the cost is consumed only if the time is equal to tg (thanks to the conditional statement) and the resulting state is an α-equivalent state. A detailed proof is in App. B.1. □

Corollary 1. *An upper bound of $\mathcal{P}_{tg}$ is a time-parametrized upper bound of $\mathcal{P}$.*

4.2 Rendering Time Explicit

Now we eliminate timing behavior from our programs. That transformation takes a program $\mathcal{P}_{tg}$ and generates $et\mathcal{P}_{tg}$ (explicit time program). The program $et\mathcal{P}_{tg}$ does not contain any timing constructs. The transformation adds a new parameter *time* to each method of the program (disjoint from all existing variable names). We transform the program in such a way that the local variable *time* of every task coincides with the global clock whenever the task is executing.

After the execution of each method the value of variable *time* is returned together with its original return value. For the technical realization we assume that future variables now store a pair of values. The original value is accessed with y.**get** and the time value with $y.time$. A method declaration

Transformed call

```
1 call(dur,tg,time){
2  while (dur>0) {
3    if(tg==time)
4        cost 1;
5    dur= dur-1;
6    release;time=max(
     time+1,time);
7   }
8  return time }
```

Transformed normalBehavior

```
9 normalBehavior(mx,dur,tc,tg,time){
10  if (time>tc && time<tc+20)
11   midnightWindow(mx,dur,tc,tg,time);
12  else{
13   ts!call(dur,tg,time);
14   release; time=max(time+1,time);
15   normalBehavior(mx,dur,tc,tg,time);
16  }
17  return time }
```

Fig. 7. Some methods of the transformed program

$T\ m(\overline{T}\ \overline{x})\{s;\textbf{return}\ p\}$ becomes $(T, Int)\ m(\overline{T}\ \overline{x}, Int\ time)\{\tau(s);\textbf{return}\ (p, time)\}$. Methods that did not return a value now return *time*.

Method statements are transformed using the function τ defined in Fig. 6. (1) Whenever a method is called, the current time is passed as a parameter. Hence, new tasks have a local time that starts when the task is created. For the **until**(p_l) (2) and **await duration**(p_l) (3) statements the lock is released and time is updated. (4) When we execute an **await** y? we compare the current local time and the time returned by the future and keep the maximum. (5) untilp(t) is an auxiliary statement used when the lock of the task has been released. We substitute it by the auxiliary instruction take generated by **release** and then update the time value. This way the transformations of **until**(p_l) and untilp(t) are coherent. Note that untilp(t) cannot appear in a program because it is an auxiliary instruction. However, by defining its transformation, we can use τ to define τ-equivalent execution states and prove the soundness of the transformation (see Def. 8 and Thm. 4.2). (6) (7) and (8) the tranformation of non-atomic statements is the transformation of the statements that compose them. (9) The remaining statements are not modified. Finally, every reference to *now*() is substituted by a reference to the new local variable *time*.

Example 3. In Fig. 7 we show some of the transformed methods. Every method has two additional parameters *tg* and *time*; the references to *now*() have been substituted by *time* (lines 3 and 10); The **await duration**(1) instructions have been replaced by a **release** followed by an update of the variable *time* (lines 6 and 14); the methods return the time (lines 8 and 17) and the cost statement in method call is wrapped into a conditional statement (line 3).

This transformation is valid for non-blocking programs. Intuitively, a program is blocking if we can have a situation where a task is waiting for the completion of another without releasing its object's lock (only possible with y.**get** instructions). In that case, other tasks in the same object could be delayed in time although they are ready to execute because they cannot access the lock. A sufficient condition to guarantee that a program is non-blocking is that every y.**get** instruction is preceded by an **await** y?. Given that condition, the task related to

y is guaranteed to be finished when y.**get** is reached and the y.**get** instruction will be completed immediately without blocking.

Definition 7 (Non-blocking program). *A program $\mathcal{P}$ is non-blocking iff for every reachable state S, if $tsk(tk, o, l, y.\textbf{get}; s) \in S$ such that $l(y) = tk_1$, then $tsk(tk_1, o_1, l_1, \epsilon(v)) \in S$.*

Similar to the previous transformation we can define a τ-equivalent execution state of S by applying the transformation to all the pending statements of all the tasks in S, obtaining their extended local state and removing the clock. The objects remain unchanged. A necessary condition for a state to be τ-equivalent is that the values of the local variables *time* in all the tasks that hold the object's lock correspond to the global clock. The tasks without the lock may have an outdated time value (smaller or equal than the global clock).

Definition 8 (τ-equivalent state). *Let S be a state of the original program with $clk(t) \in S$. Then a τ-equivalent state S^τ to S is defined as:*
$S^\tau = \{ob(o, a, lk) \mid ob(o, a, lk) \in S\} \cup$
$\quad \{tsk(tk, o, l + [time \to T_{tk}], \{\tau(s); \textbf{return } (p, time)\})$
$\quad\quad \mid tsk(tk, o, l, \{s; \textbf{return } p\}) \in S\}$
where T_{tk} represents the current time of each task. We require that $T_{tk} \leq t$ for all tasks and $T_{tk} = t$ for the tasks that have the lock.

Theorem 2 ($et\mathcal{P}_{tg}$ Simulates $\mathcal{P}_{tg}$). *Given a non-blocking program $\mathcal{P}_{tg}$ and its transformed program $et\mathcal{P}_{tg}$, then for every trace $tr(\overline{val}) \in \mathcal{TR}_{\mathcal{P}_{tg}}$ with states $S_{1..n}$ there is a trace $tr'(\overline{val}, 1) \in \mathcal{TR}_{et\mathcal{P}_{tg}}$ that contains the τ-equivalent states $S^\tau_{1..n}$, in the same order, S^τ_n being the final state of tr' and $Cost_{tr(\overline{val})}(tg) = Cost_{tr'(\overline{val},1)}$. Note that tr' can have intermediate states that do not have an equivalent in tr. Additionally, consecutive states in tr might have a single τ-equivalent state in tr'.*

Proof idea By induction on the length of a trace. The base case is trivial. In the inductive step, we assume that we have a trace $tr \in \mathcal{TR}_{\mathcal{P}_{tg}}$ of length n and a trace $tr' \in \mathcal{TR}_{et\mathcal{P}_{tg}}$ such that tr' contains n τ-equivalent states to the ones appearing in tr in the same order (tr' can have additional intermediate states). The cost of tr and tr' is equal. We prove that for any step from the final state of tr $S \xrightarrow{c} S_{new}$, there is a step or sequence of steps in the transformed program that reaches a τ-equivalent state with the same resource consumption.

For the rules that are applied to a task with a lock and that are not affected by τ we apply the same rule to the transformed program and obtain the desired state. The fact that the program is non-blocking implies that when we apply (Tick), all the locks must be free. If we apply (Tick) in $\mathcal{P}_{tg}$, all the local variables *time* become outdated but as the tasks do not have the lock, the state in tr' is still τ-equivalent. For the rules (Await1), (Until1), (Until2), and (Until3) we have to prove that the assigned time in tr' corresponds to the clock in tr. The idea is that since the rules are applicable (the awaited task is finished or the awaited time is reached), the clock cannot advance before they are applied (this is only

valid for non-blocking programs). Therefore, $max(time, v.time)$, $max(p_l, time)$ and $max(t, time)$ yield the correct value of the clock and the reached state in tr' is τ-equivalent. Similarly, since the rule (ACTIVATE) is applicable (a **release** or task creation), the clock cannot have advanced and the variable *time* is not outdated. The statement **await duration**(d) can be reduced to **until**$(t + d)$. A detailed proof can be found in App. B.2. □

Corollary 2. *An upper bound of $et\mathcal{P}_{tg}$ is an upper bound of $\mathcal{P}_{tg}$.*

Note that theorem states the soundness of the transformation but not its completeness. The transformation is in fact not complete. That is, the transformed program can have additional behaviors that are not present in the original program. In particular, if we have an **until**(p_l) statement, it is possible that when executed $eval(p_l, l) \leq t$ and the semantic rule UNTIL1 is applied. This rule does not release the lock. However, the corresponding transformed program will always release the lock $\tau($**until**$(p_l)) = release; time = max(p_l, time);$. The transformed program could then schedule another pending task whereas the original program cannot do the same. However, we do not expect any additional loss of precision for this reason because the later analyses already conservatively approximate a possible release of the lock in any instruction that contains a conditional release (such as **await** y?).

5 Analyzing Untimed Programs

We are able to apply existing tools for resource analysis on transformed programs to obtain an upper bound or a set of upper bounds, but it is crucial to ensure high precision of the analysis (cf. Fig. 4). To achieve this we combine the frontend of the resource analyzer COSTABS [1] with the more recent solver CoFloCo[1] [10]. The input of COSTABS is an untimed ABS program (i.e., a superset of the language in Sect. 2) and generates a set of *cost equations* that characterize the cost of the program and can be solved by CoFloCo. Cost equations are non-deterministic recurrence equations annotated with constraints.

Example 4. COSTABS+CoFloCo generates 15 conditional upper bounds for the main method of our example. We show some of them, the complete data from the example can be found in App. A.

#	Upper Bound	Precondition
6	tg	$(mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1 \wedge tc \geq tg + 1)$
8	$mx + tc$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc+2 \wedge dur \geq tg+1 \wedge tc+18 \geq tg) \vee (dur = tg \wedge mx \geq 1 \wedge tc \geq 2 \wedge tc + 18 \geq dur \wedge dur \geq tc + 2) \vee (tg = tc + 19 \wedge mx \geq 1 \wedge tg \geq 20 \wedge dur \geq tg + 1) \vee (tg = tc + 1 \wedge mx \geq 1 \wedge tg \geq 2 \wedge dur \geq tg + 1)$
12	$tg - tc - 19$	$(mx \geq 1 \wedge tc \geq 1 \wedge tg \geq tc+21 \wedge tg \geq dur+tc \wedge dur+tc+18 \geq tg)$
15	$mx + dur + tc - tg$	$(mx \geq 1 \wedge tg \geq dur+1 \wedge tg \geq tc+2 \wedge tc+18 \geq tg \wedge dur+tc \geq tg+2)$

[1] Other resource analysis tools could be used as long as they admit a parametric cost model, i.e., it is must be possible to establish the cost at each program point.

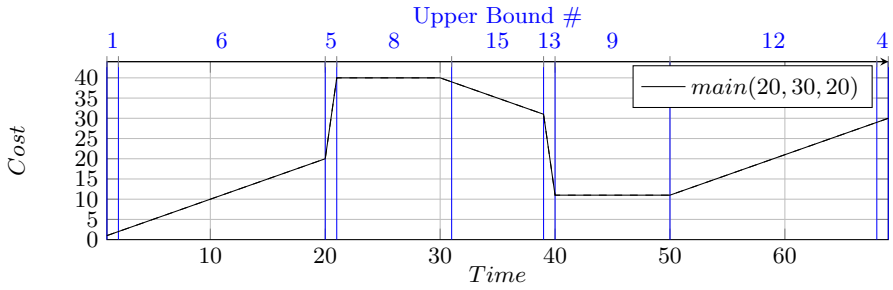


Fig. 8. A resource consumption profile generated to observe other upper bounds

```
1Unit m(p,q,r,s,n){
2 await duration(nondet);
3 this!r(r);
4 while(n>0){
5  this!p(p);
6  y=this!q(q);
7  await y?;
8  await duration(nondet);
9  await duration(1);
10  n=n-1;
11 } this!s(s);
12}
```

```
13Unit r/p/q/s(c){
14 await duration(nondet);
15 cost c;
16}
```

#	Upper bound	Precondition
1	$max(s,q)+r+n*p$	$n \geq 3 \wedge tg \geq n$
2	$q+r+(tg+1)*p$	$tg \geq 2 \wedge n \geq tg+2$

Fig. 9. A timed model annotated to obtain peak cost

In Fig. 4, we could observe the precision of the upper bounds that correspond to the resource consumption in simulations for concrete inputs. By examining the upper bounds, we can get a good approximation of the different behaviors of the program. Preconditions can be useful to see under which conditions a program behaves differently. In particular, we can generate test data from preconditions that generate a certain behavior. We used this idea to create execution patterns of our example that were not observed in the initial simulations, for example upper bound #15 in Fig. 8. The upper bounds here are also tight and the intervals where each upper bound is valid are annotated with its number.

6 Inferring Peak Cost

In Fig. 9, we introduce a new example to illustrate how explicit time primitives can be used for the purpose of modelling peak cost. The example is an adaptation of the running example from [2]. In the original example, the methods r,p,q and s are left unspecified. We assume they have all the same implementation and consume the amount c of resources which is given by the input parameter. With peak cost we mean the following: at any time tg there is a set of pending tasks that could be executed simultaneously. Let $C(tg)$ be the sum of the cost of these

tasks at tg. Then the peak cost is the maximum value that $C(tg)$ attains. In [2] the notion of simultaneity is defined based on synchronization points **await** y? (line 7). In order to emulate the behavior that corresponds to peak cost analysis, we simply make time advance one unit right after each synchronization point (line 9 in gray). In addition, with this concept of simultaneity, tasks do not need to start executing immediately but can be delayed an indefinite amount of time. We can model that by inserting a non-deterministic **await duration**($nondet$) whenever a method starts or restarts to execute (lines 2, 8 and 14).

After adding these annotations we apply our analysis and obtain upper bounds of the peak cost. On the right side of Fig. 9 we display the two main conditional upper bounds (we leave out some border cases) computed by our analysis. The first one corresponds to the result obtained by [2] and captures that there can be n pending instances of task p to be executed simultaneously, but only one instance of q as the program waits for each instance to finish before calling the next one. This is the peak cost for executing the complete program as reflected in the precondition $tg \geq n$. The second upper bound gives more fine-grained information on how the tasks might accumulate in the loop as time tg passes. This kind of analysis cannot be obtained with the method of [2], because it involves analyzing timed behavior.

7 Related Work, Conclusions and Future Work

To the best of our knowledge, we present the first resource analysis for timed concurrent object-oriented programs. The analysis is based on an inexpensive program transformation into untimed programs. The untimed programs can be analyzed with existing tools for resource analysis [7, 8, 10, 11, 15] so our analysis directly benefits from any improvement of these techniques. Because the timing behavior is parametric, our analysis opens multiple possibilities for reasoning about concurrent programs. For instance, we could generate timing annotations according to a network model and observe how processing power is consumed.

The most closely related work is about resource analysis of concurrent objects [1, 2]. The main focus of [1] is on how to deal with shared memory in concurrent applications and they propose the notion of *cost centers* as a way of computing the cost of different components (objects) separately. In contrast, our approach analyzes cost over time—it can be seen as a layer on top of their analysis. As mentioned in Sec. 6, paper [2] does not measure total cost but *peak cost*, i.e., the maximum cost a component can consume at any given time. The crucial difference is that in our analysis the cost is determined by the explicit time behavior, whereas [2] abstracts away from timed behavior and uses the maximal degree of parallelism obtained from a *May-Happen-in-Parallel* analysis [3]. Because we have explicit time primitives, we can infer upper bounds that depend on time. Additionally, we can infer peak cost parametrized with time (see Sec. 6). The authors of paper [12] define a type system to infer upper bounds on two cost models (*work* and *depth*) for functional programs. *work* is the total cost of the

program and *depth* is similar to the maximum time that can be reached in any part of the program if we make time advance each time resources are consumed.

Timed automata [4] have been widely used in the analysis of timed systems. In particular, they have been applied to concurrent object-oriented programs [9], however, with a different focus: to transform such programs into timed automata to check their schedulability. *Priced timed automata* [5] model systems with both time and resources, but they differ from our setting in crucial ways, having continuous time and no input parameters. They also focus on different properties, such as *minimum-cost reachability*.

Our current analysis does not support blocking programs. When we have a blocking program, there can be extra delays in tasks after instructions that release an object's lock (**await** $y?$, **release**, and **await duration**(t)). These extra delays depend on the blocking instructions that might interleave with such tasks and they must be taken into account to update the *time* variable correctly. How to approximate blocking programs safely and precisely is left to future work.

References

1. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *P*roc. of APLAS'11, volume 7078 of *L*NCS, pages 238–254. Springer, Dec. 2011.
2. E. Albert, J. Correas, and G. Román-Díez. Peak Cost Analysis of Distributed Systems. In *2*1st International Static Analysis Symposium (SAS'14), volume 8723 of *L*NCS, pages 18–33. Springer-Verlag, 2014.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of may-happen-in-parallel in concurrent objects. In H. Giese and G. Rosu, editors, *F*ORTE, volume 7273 of *L*NCS, pages 35–51. Springer-Verlag, 2012.
4. R. Alur and D. L. Dill. A theory of timed automata. *T*heoretical Computer Science, 126:183–235, 1994.
5. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced time automata. In M. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *H*SCC, volume 2034 of *L*NCS, pages 147–161. Springer, 2001.
6. J. Bjørk, F. Boer, E. Johnsen, R. Schlatte, and S. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *I*nnov. Syst. Softw. Eng., 9(1):29–43, Mar. 2013.
7. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *T*ACAS, 2014.
8. Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *P*LDI, 2015. To Appear.
9. F. de Boer, T. Chothia, and M. Jaghoori. Modular schedulability analysis of concurrent objects in creol. In F. Arbab and M. Sirjani, editors, *F*SEN, volume 5961 of *L*NCS, pages 212–227. Springer, 2010.
10. A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In J. Garrigue, editor, *A*PLAS, volume 8858 of *L*NCS, pages 275–295. Springer, Nov. 2014.
11. S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *P*OPL, pages 127–139, New York, NY, USA, 2009. ACM.

12. J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In *E*SOP, 2015. To Appear.
13. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *P*roc. of FMCO 2010, volume 6957 of *L*NCS, pages 142–164. Springer-Verlag, 2011.
14. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *J*ournal of Logical and Algebraic Methods in Programming, 2014. DOI: 10.1016/j.jlamp.2014.07.001.
15. M. Sinn, F. Zuleger, and H. Veith. A simple and scalable approach to bound analysis and amortized complexity analysis. In *C*AV, volume 8559 of *L*NCS, pages 743–759. Springer, 2014.

Appendix J

Article *Resource analysis of complex programs with cost equations*, [18]

Resource Analysis of Complex Programs with Cost Equations

Antonio Flores-Montoya and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science
aflores|haehnle@cs.tu-darmstadt.de

Abstract. We present a novel static analysis for inferring precise complexity bounds of imperative and recursive programs. The analysis operates on cost equations. Therefore, it permits uniform treatment of loops and recursive procedures. The analysis is able to provide precise upper bounds for programs with complex execution flow and multi-dimensional ranking functions. In a first phase, a combination of control-flow refinement and invariant generation creates a representation of the possible behaviors of a (possibly inter-procedural) program in the form of a set of execution patterns. In a second phase, a cost upper bound of each pattern is obtained by combining individual costs of code fragments. Our technique is able to detect dependencies between different pieces of code and hence to compute a precise upper bounds for a given program. A prototype has been implemented and evaluated to demonstrate the effectiveness of the approach.

1 Introduction

Automatic resource analysis of programs has been subject to intensive research in recent years. This interest has been fuelled by important advances in termination proving, including not only ranking function inference [6, 16], but complete frameworks that can efficiently prove termination of complex programs [3, 7, 10]. Termination proving is, however, only one aspect of resource bound inference.

There are several approaches to obtain upper bounds for imperative programs [3, 8, 9, 12–15, 17, 18]. Most pay little attention to interprocedural, in particular, to recursive programs. Only SPEED [14] and the recent paper [8] address recursive procedures. The extent to which SPEED can deal with complex recursive procedures is hard to evaluate (they provide only one example). The approach of [8] ignores the output of recursive calls which, however, can be essential to obtain precise bounds (see Fig.1).

A different line of work is based on *Cost Equations*, a particular kind of *non-deterministic recurrence relations*, annotated with constraints. This is the approach followed by the COSTA group [1, 2, 4, 5]. One advantage of Cost Equations is that they can deal with both loops and recursion in a *uniform* manner. However, the approach does not cope well with loops that exhibit multiple phases or with programs whose termination proof requires multiple linear ranking functions for a single loop/recursive procedure.

We use the program in Fig.1 to illustrate some of the problems we address in this paper. The program is annotated with structured comments containing cost labels of the form $[Cost\ x]$. These indicate that at the given program point x resource units are consumed. The program consists of two methods. Method `move` behaves differently depending on the value of boolean variable `fwd`. If `fwd` is true, it may call itself recursively with $n' = n + 1$ and consume two resource units. If `fwd` is false, it may call itself with $n' = n - 1$ and consume one resource unit. Method `main` has a loop that calls `move` and updates the value of n with the result of the call. Additionally, at any iteration, it can change the value of `fwd` to true.

Program 1

```
1 main(int m, int n){
2  //assume(m>n>0)
3  bool fwd=false;
4  while(n > 0){
5    n=move(n,m,fwd);
6    if(?) fwd=true;
7  }
8 }
9 int move(int n,m, bool fwd){
10  if(fwd){
11   if(m > n && ?){
12     ...; //[Cost 2]
13     return move(n+1,m,fwd);
14   }
15  }else{
16   if(n > 0 && ?){
17    ...; //[Cost 1]
18    return move(n-1,m,fwd);
19   }
20  }
21  return n;
22 }
```

Fig. 1. Program example

This example is challenging for several reasons: (i) `move` behaves differently depending on the value of `fwd`, so we ought to analyse its different behaviors separately; (ii) the return value of `move` influences the subsequent behavior of the `main` method and has to be taken into account; (iii) the `main` method might not terminate and yet its cost is finite. Moreover, the upper bound of terminating and non-terminating executions is different. Below we present a table that summarizes the possible upper bounds of this program.

Execution pattern	(1)	(2)	(3)
Upper bound	$n + 2m$	$2(m - n)$	n
Terminating	×	×	✓

Pattern (1) occurs when `move` decrements `n` for a while but without reaching 0 (the initial `n` is an upper bound of the cost); then the guard in line 6 is true and `move` increases `n` up to `m`, incurring a cost of $2m$. The loop in `main` never terminates because n does not reach 0. In pattern (2) the guard in line 6 is true at the beginning and `move` increases `n` to `m` consuming $2 * (m - n)$. Finally, in pattern (3), the guard in line 6 is never true (or only when $n = 0$). Then `move` decrements `n` to 0 and the main loop may terminate, consuming n resource units.

The techniques presented in our paper can deal fully automatically with complex examples such as the program above. Our main contributions are: first, a static analysis for both imperative and (linearly) recursive programs that can infer precise upper bounds for programs with complex execution patterns as above. The analysis combines a control-flow refinement technique in the abstract context of cost equations and a novel upper bound inference algorithm. The latter exploits dependencies between different parts of a program during the computa-

tion of upper bounds and it takes into account multiple upper bound candidates at the same time. Second, we provide an implementation of our approach. It is publicly available (see Sec. 6) and it has been evaluated in comparison with KoAT [8], PUBS [1] and Loopus[17]. The experimental evaluation shows how the analysis deals with most examples presented as challenging in the literature.

2 Cost Equations

In this section, we introduce the necessary concepts for the reasoning with cost equations. The symbol $\overline{x}$ represents a sequence of variables $x_1, x_2, \cdots, x_n$ of any length. The expression $vars(t)$ denotes the set of variables in a generic term t. A variable assignment $\alpha : V \mapsto D$ maps variables from the set of variables V to elements of a domain D and $\alpha(t)$ denotes the replacement of each $x \in vars(t)$ by $\alpha(x)$. A *linear expression* has the form $q_0 + q_1 * x_1 + \cdots + q_n * x_n$ where $q_i \in \mathbb{Q}$ and $x_1, x_2, \cdots, x_n$ are variables. A *linear constraint* is $l_1 \leq l_2$, $l_1 = l_2$ or $l_1 < l_2$, where l_1 and l_2 are linear expressions. A *cost constraint* φ is a conjunction of linear constraints $l_1 \wedge l_2 \wedge \cdots \wedge l_n$. The expression $\varphi(\bar{x})$ represents a cost constraint φ instantiated with the variables $\bar{x}$. A cost constraint φ is *satisfiable* if there exists an assignment $\alpha : V \mapsto \mathbb{Z}$ such that $\alpha(\varphi)$ is valid (α satisfies φ).

Definition 1 (Cost expression). *A* cost expression e *is defined as:*

$$e ::= q \mid nat(l) \mid e + e \mid e * e \mid nat(e - e) \mid \max(S) \mid \min(S)$$

where $q \in \mathbb{Q}^+$, l is a linear expression, S is a non-empty set of cost expressions and $nat(e) = \max(e, 0)$. We often omit $nat()$ wrappings in the examples.

Definition 2 (Cost equation). *A* cost equation c *has the form $\langle C(\overline{x}) = e + \sum_{i=1}^{n} D_i(\overline{y_i}), \varphi\rangle$ ($n \geq 0$), where C and D_i are cost relation symbols; all variables $\overline{x}$, $\overline{y_i}$, and $vars(e)$ are distinct; e is a cost expression; and φ is a conjunction of linear constraints that relate the variables of c.*

A cost equation $\langle C(\overline{x}) = e + \sum_{i=1}^{n} D_i(\overline{y_i}), \varphi\rangle$ states that the cost of $C(\overline{x})$ is e plus the sum of the costs of each $D_i(\overline{y_i})$. The relation φ serves two purposes: it restricts the applicability of the equation with respect to the input variables and it relates the variables $\overline{x}$, $vars(e)$, and $\overline{y_i}$. One can view C as a non-deterministic procedure that calls $D_1, D_2, \ldots, D_n$.

Fig. 2 displays the cost equations corresponding to the program in Fig. 1. To simplify presentation in the examples we reuse some variables in different relation symbols. In the implementation they are in fact different variables with suitable equality constraints in φ.

We restrict ourselves to linear recursion, i.e., we do not allow recursive equations with more than one recursive call. Our approach could be combined with existing analyses for multiple recursion such as the one in [4]. Input and output variables are both included in the cost equations and treated without distinction. By convention, output variable names end with "o" so they can be easily recognized. In a procedure, the output variable corresponds to the return variable (*no*

SCC	Nr	*Cost Equation*
S_1	1	$main(n,m) = while(n,m,0) \quad n \geq 1 \wedge m \geq n+1$
S_2	2	$while(n,m,fwd) = 0 \quad n \leq 0$
	3	$while(n,m,fwd) = move(n,m,fwd,no) + while(no,m,fwd) \quad n > 0$
	4	$while(n,m,fwd) = move(n,m,fwd,no) + while(no,m,1) \quad n > 0$
S_3	5	$move(n,m,fwd,no) = 2 + move(n+1,m,fwd,no) \quad fwd = 1 \wedge n < m$
	6	$move(n,m,fwd,no) = 0 \quad fwd = 1 \wedge n = no$
	7	$move(n,m,fwd,no) = 1 + move(n-1,m,fwd,no) \quad fwd = 0 \wedge n > 1$
	8	$move(n,m,fwd,no) = 0 \quad fwd = 0 \wedge n = no$

Fig. 2. Cost equations of the example program from Fig. 1

in the method move). In a loop, the output variables are the local variables that might be modified inside the loop. In the while loop from Fig.2, we would have $while(n,m,fwd,no,fwdo)$ where no and $fwdo$ are the final values of n and fwd, but the cost equations have been simplified for better readability.

Generating Cost Equations Cost equations can be generated from source code or low level representations. Loop extraction and partial evaluation are combined to produce a set of cost equations with only direct recursion [1]. The details are in the cited papers and omitted for lack of space. The resulting system is a sequence of strongly connected components (SCCs) $S_1, \ldots, S_n$ such that each S_i is a set of cost equations of the form $\langle C(\overline{x}) = e + \sum_{j=1}^{k} D_j(\overline{y_j}) + \sum_{j=1}^{n} C(\overline{y_j}), \varphi \rangle$ with $k \geq 0$ and $n \in \{0,1\}$ and each $D_j \in S_{i'}$ where $i' > i$. Each SCC is a set of directly recursive equations with at most one recursive call and k calls to SCCs that appear later in the sequence. Hence, S_1 is the outermost SCC and entry point of execution while S_n is the innermost SCC and has no calls to other SCCs. Each resulting cost equation is a complete iteration of a loop or recursive procedure.

Example 1. In Fig. 2, the cost equations of Program 1 are grouped by SCC. Each SCC defines only one cost relation symbol: *main*, *while*, and *move* occur in S_1, S_2, and S_3, respectively. However, the cost equations in any SCC may contain references to equations that appear later. For instance, equations 3 and 4 in S_2 have references to *move* in S_3.

A concrete execution of a relation symbol C in a set of cost equations is generally defined as a (possibly infinite) evaluation tree $T = node(r, \{T_1, \ldots T_n\})$, where $r \in \mathbb{R}^+$ is the cost of the root (an instance of the cost expression in C) and $T_1, \ldots T_n$ are sub-trees corresponding to the calls in C. In the following we will not need this general definition. A formal definition of evaluation trees and their semantics is in [1].

3 Control-flow Refinement of Cost Equations

As noted in Sec. 1, we have to generate all possible execution patterns and discard unfeasible patterns that might reduce precision or even prevent us from

obtaining an upper bound. Our cost equation representation allows us to look at one SCC at a time. If we consider only the cost equations within one SCC, we have sequences of calls instead of trees (we are only considering SCCs with linear recursion). That does not prevent each cost equation in the sequence from having calls to other SCCs.

Example 2. Given S_3 from Fig. 2, the sequence $5 \cdot 5 \cdot 6$ represents a feasible execution where equation 5 is executed twice followed by one execution of 6. On the other hand, the execution $5 \cdot 8$ is infeasible, because the cost constraints of its elements are incompatible ($fwd = 1$ and $fwd = 0$).

Given an SCC C consisting of cost equations S_C, we can represent its execution patterns as regular expressions over the alphabet of cost equations in S_C. We use a specific form of execution patterns that we call *chain*:

Definition 3 (Phase, Chain). *Let $S_C = c_1, \ldots, c_r$ be the cost equations of an SCC C. A* phase *is a regular expression $(c_{i_1} \vee \ldots \vee c_{i_m})^+$ over S_C (executed a positive number of times). A special case is a phase where exactly one equation is executed: $(c_{i_1} \vee \ldots \vee c_{i_m})$.*

A chain *is a regular expression over S_C composed of a sequence of phases $ch = ph_1 \cdot ph_2 \cdots ph_n$ such that its phases do not share any common equation. That is, if $c \in ph_i$, then $c \notin ph_j$ for all $j \neq i$.*

We say that a cost equation that has a recursive call is *iterative* and a cost equation with no recursive calls is *final*. Given an SCC C consisting of cost equations S_C, we use the name convention $i_1, i_2 \ldots i_n$ for the iterative equations and $f_1, f_2 \ldots f_m$ for the final equations in S_C. All possible executions of an SCC can be summarized in three basic chains: (1) $ch_n = (i_1 \vee i_2 \vee \cdots \vee i_n)^+ \cdot (f_1 \vee f_2 \vee \cdots \vee f_m)$ an arbitrary sequence of iterations that terminates with one of the base cases; (2) $ch_b = (f_1 \vee f_2 \vee \cdots \vee f_m)$ a base case without previous iterations; (3) an arbitrary sequence of iterations that never terminates $ch_i = (i_1 \vee i_2 \vee \cdots \vee i_n)^+$.

Example 3. The basic chains of method *move* (SCC S_3 of Fig.2) are: $ch_n = (5 \vee 7)^+(6 \vee 8)$, $ch_b = (6 \vee 8)$ and $ch_i = (5 \vee 7)^+$. Obviously, these chains include a lot of unfeasible call sequences which we want to exclude.

3.1 Chain Refinement of an SCC

Our objective is to specialize a chain into more refined ones according to the constraints φ of its cost equations. To this end, we need to analyse the possible sequences of phases in a chain. We use the notation $c \in ch$ to denote that the cost equation c appears in the chain ch.

Definition 4 (Dependency). *Let $c, d \in ch$, $c = \langle C(\bar{x}_c) = \ldots + C(\bar{z}), \varphi_c \rangle$, $d = \langle C(\bar{x}_d) = \ldots, \varphi_d \rangle$; then $c \preceq d$ iff the constraint $\varphi_c \wedge \varphi_d \wedge (\bar{z} = \bar{x}_d)$ is satisfiable. Intuitively, $c \preceq d$ iff d can be executed immediately after c. The relation $\preceq^*$ is the transitive closure of $\preceq$.*

We generate new phases and chains according to these dependencies. Define $c \equiv d$ iff $c = d$ (syntactic equality) or $c \preceq^* d$ and $d \preceq^* c$. Each equivalence class in $[c]_\equiv$ gives rise to a new phase. If $[c]_\equiv = \{c\}$ and $c \not\preceq c$, the new phase is (c). If $[c]_\equiv = \{c_1, \ldots, c_n\}$, the new phase is $(c_1 \vee \cdots \vee c_n)^+$. To simplify notation we identify an equivalence class with the phase it generates. Then $ph \prec ph'$ iff $ph \neq ph'$, $c \in ph$, $d \in ph'$ and $c \preceq d$. $ch' = ph_1 \cdots ph_n$ is a *valid chain* iff for all $1 \leq i < n$: $ph_i \prec ph_{i+1}$.

Example 4. The dependency relation of *move* (SCC S_3 from Fig. 2) is the following: $5 \preceq 5$, $5 \preceq 6$, $7 \preceq 7$ and $7 \preceq 8$. This produces the following phases: $(5)^+$, $(7)^+$, (6) and (8), which in turn give rise to chains: non-terminating chains $(5)^+$, $(7)^+$; terminating chains $(5)^+(6)$, $(7)^+(8)$ and the base cases (6), (8). This refinement captures the important fact that the method cannot alternate the behavior that increases n (cost equation 5) with the one that decreases it (cost equation 7).

Theorem 1 (Refinement completeness). *Let $ch_1, \ldots, ch_n$ be the generated chains for a SCC S from the basic chains of S. Any possible sequence of cost equation applications of S is covered by at least one chain ch_i, $i \in 1..n$ (a proof can be found in [11]).*

3.2 Forward and Backward Invariants

We can use invariants to improve the precision of the inferred dependencies and to discard unfeasible execution patterns. Given a chain $ch = ph_1 \cdots ph_n$ in S_i with C as cost relation symbol, we can infer forward invariants (*fwdInv*) that propagate the context in which the chain is called from ph_1 to the subsequent phases. Additionally, we can propagate the relation between the variables from the final phase ph_n to the previous phases until calling point ph_1, obtaining backward invariants (*backInv*). These invariants provide us with extra information at each phase ph_i coming from the phases that appear before (*fwdInv*) or after (*backInv*) ph_i.

$fwdInv_{ch}(ph_i)$ and $backInv_{ch}(ph_i)$ denote forward and backward invariants valid at any application of the equations in the phase ph_i of chain ch. If it is obvious which chain is referred to, we leave out the subscript ch. The forward invariant at the beginning of a chain ch in an SCC S_i is given by the conditions under which ch is called in other SCCs. The backward invariant at the end of a chain ch is defined by the constraints φ of the base case ph_n for terminating chains. For non-terminating chains, the backward invariant at the end of a chain is the empty set of constraints (*true*). The backward invariant of the first phase of a chain ch represents the input-output relations between the variables. It can be seen as a summary of the behavior of ch. The procedure for computing these invariants can be found in [11].

Additionally, we define φ_{ph} and φ_{ph^*} for iterative phases. The symbol φ_{ph} represents the relation between the variables before and after any positive number of iterations of ph, while φ_{ph^*} represents the relation between the variables before and after zero or more iterations.

Example 5. Some of the inferred invariants for the chains of S_3 of our example:
$backInv_{(5)^+(6)}((5)^+) = fwd = 1 \wedge m > n \wedge m \geq no \wedge no > n$
$backInv_{(7)^+(8)}((7)^+) = fwd = 0 \wedge n > 0 \wedge no \geq 0 \wedge n > no$
These invariants reflect applicability conditions (Such as $fwd = 0$) and the relation between the input and the output variables. For example, $no > n$ holds when n is increased and $n > no$ when it is decreased. The condition $m \geq no$ is derived from the fact that at the end of phase $(5)^+$ we have $m > n$, in phase (6) $n' = no'$ and the transition is $n' = n + 1 \wedge no' = no$.

We can use forward and backward invariants to improve the precision of the inferred dependencies. At the same time, a more refined set of chains will allow us to infer more precise invariants. Hence, we can iterate this process (chain refinement and invariant generation) until no more precision is achieved or until we reach a compromise between precision and performance. We can also use the inferred invariants to discard additional cost equations or chains. Let $c = \langle C(\bar{x}) = \ldots + C(\bar{z}), \varphi \rangle \in ph_i$, if $\varphi \wedge backInv_{ch}(\overline{ph_i}) \wedge fwdInv_{ch}(ph_i)$ is unsatisfiable, c cannot occur and can be eliminated from ph_i in the chain ch. If any invariant belonging to a chain is unsatisfiable its pattern of execution cannot possibly occur and the chain can be discarded.

3.3 Terminating Non-termination

In our refinement procedure, we distinguish terminating and non-terminating chains explicitly. Given a chain $ph_1 \cdots ph_n$, it is assumed that every phase ph_i with $i \in 1..n-1$ is terminating. This is safe, because for each ph_i that is iterative we generated another chain of the form $ph_1 \cdots ph_i$, where ph_i is assumed not to terminate. That is, we consider both the case when ph_i terminates and when it does not terminate. Given a non-terminating chain, if we prove termination of its final phase, we can safely discard that chain.

Consider a phase $(c_1 \vee c_2 \vee \ldots \vee c_m)^+$, we obtain a (possibly empty) set of linear ranking functions for each c_i, denoted RF_i, using the techniques of [16, 6]. A linear ranking function of a cost equation $\langle C(\overline{x}) = \cdots + C(\overline{x}'), \varphi \rangle$ with a recursive call $C(\overline{x}')$ is a linear expression f such that (1) $\varphi \Rightarrow f(\overline{x}) \geq 0$ and (2) $\varphi \Rightarrow f(\overline{x}) - f(\overline{x}') \geq 1$.

For each ranking function f of c_i, we check whether its value can be incremented in any other $c_j = \langle C(\overline{x}) = \cdots + C(\overline{x}'), \varphi_j \rangle$, $j \neq i$ (whether $\varphi_j \wedge f(\overline{x}) - f(\overline{x}') < 0$ is satisfiable). If f can be increased in c_j we say that f depends on c_j. As in [3], the procedure for proving termination consists in eliminating the cost equations that have a ranking function without dependencies first. Then, incrementally eliminate the cost equations that have ranking functions whose dependencies have been already removed until there are no cost equations remaining. The set of ranking functions and their dependencies will be used again later to introduce specific bounds for the number of calls to each c_i.

Example 6. The ranking functions for the phases $(5)^+$ and $(7)^+$ are $m - n$ and n respectively. With such ranking functions, we can discard the non-terminating chains $(5)^+$ and $(7)^+$. The remaining chains are $(5)^+(6)$, $(7)^+(8)$, (7) and (8).

Nr	*Cost Equation*
3.1	$while(n,m,fwd) = move_{(5)^+(6)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd} = \mathbf{1} \wedge \mathbf{m} > \mathbf{n} \wedge \mathbf{m} \geq \mathbf{no} \wedge \mathbf{no} > \mathbf{n}$
3.2	$while(n,m,fwd) = move_{(6)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd} = \mathbf{1} \wedge \mathbf{no} = \mathbf{n}$
3.3	$while(n,m,fwd) = move_{(7)^+(8)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd} = \mathbf{0} \wedge \mathbf{no} \geq \mathbf{0} \wedge \mathbf{n} > \mathbf{no}$
3.4	$while(n,m,fwd) = move_{(8)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd} = \mathbf{0} \wedge \mathbf{n} = \mathbf{no}$

Fig. 3. Refinement of Cost equation 3 from Fig. 2

3.4 Propagating Refinements

The refinement of an SCC S_i in a sequence $S_1, \ldots, S_n$ can affect both predecessors and successors of S_i. The initial forward invariants from SCCs that are called in S_i, the forward invariants of the SCCs $S_{i+1}, \ldots, S_n$ might be strengthened by the refinement of S_i. The preceding SCCs that have calls to S_i can be specialized so they call the refined chains. The backward invariants can be included in the calling cost equations thus introducing a "loop summary" of S_i's behavior.

Each cost equation containing a call to S_i, say $\langle D(\bar{x}) = \ldots + C_{ch}(\bar{z}), \varphi \rangle \in S_j$ with $j < i$, can be replaced with a set of cost equations $\langle D(\bar{x}) = \ldots + C_{ch'}(\bar{z}), \varphi' \rangle$, where $ch' = ph_1 ph_2 \cdots ph_m$ is one of the refined chains of ch, and $\varphi' := \varphi \wedge backInv_{ch'}(\overline{ph_1})$. If φ' is unsatisfiable, the cost equation can be discarded.

Example 7. We propagate the refinement of method *move* (SCC S_3) to *while* (SCC S_2). Fig. 3 shows how cost equation 3 is refined by substituting the calls to *move* by calls to specific chains of *move* and by adding the backward invariants of the callees to its cost constraint φ. Analogously, cost equation 4 is refined into 4.1, 4.2, 4.3, and 4.4. The only difference is that the latter have a recursive call to *while* with $fwd = 1$. The cost equations of *move* are not changed because the do not have calls to other SCCs.

The new phases are $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$, $(3.3 \vee 3.4)^+$, (4.3), (4.4) and (2). Phase $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$ represents iterations of the loop when $fwd = 1$. The fact that fwd is explicitly set to 1 in 4.1 and 4.2 does not have any effect. Phase $(3.3 \vee 3.4)^+$ represents the iterations when $fwd = 0$ and is kept that way in the recursive call. Finally, (4.3) and (4.4) are the cases where fwd is changed from 0 to 1. If we use the initial forward invariant $n \geq 1 \wedge m > n$ of *main* (in SCC S_1), we obtain the following chains:

Pattern (1)	Pattern (2)	Pattern (3)
$(3.3 \vee 3.4)^+(4.3)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(4.3)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(3.3 \vee 3.4)^+(2)$
$(3.3 \vee 3.4)^+(4.4)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(4.4)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(3.3 \vee 3.4)^+$

They are grouped according to the execution patterns that were intuitively presented in Sec. 1. Note that neither $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$ or $(3.3 \vee 3.4)^+$ are

always terminating as we can iterate indefinitely on 3.2, 4.2 and 3.4. These cases correspond to a call to *move* that immediately returns without modifying n. Therefore, we cannot discard any of the non-terminating chains.

4 Upper Bound Computation

4.1 Cost Structures

At this point, a refined program consists of a sequence of SCCs $S_1, \ldots, S_n$ where each SCC S_i contains a set of chains. We want to infer safe upper bounds for each chain individually but, at the same time, take their dependencies into account. The standard approach on cost equations [1] consists in obtaining a cost expression that represents the cost of each SCC S_i and substituting any call to that S_i by the inferred cost expression. That way, we can infer closed-form upper bounds for all SCCs in a bottom up approach (From S_n to S_1). This approach turns out not to be adequate to exploit the dependencies between different parts of the code as we illustrate in the next example.

Example 8. Let us obtain an upper bound for method *main* when it behaves as in chain $(3.3 \vee 3.4)^+(2)$. This is a simple pattern, where *move* only increases or leaves n unchanged. Following the standard approach, we first obtain the upper bound for *move* when called in 3.3 and 3.4, that is, when *move* behaves as in $(7)^+(8)$ and (8). By multiplying the maximum number of recursive calls with the maximum cost of each call the upper bound we obtain is n and 0, respectively. The cost of $(3.3 \vee 3.4)^+(2)$ is then the maximum cost of each iteration n multiplied by the maximum number of iterations. However, 3.4 can iterate indefinitely, so we fail to obtain an upper bound.

If we apply the improved method of [4] after the refinement, we consider 3.3 and 3.4 independently. Phase 3.3 has zero cost and 3.4 has a ranking function n, yielding a bound of n^2 for this chain (while a more precise bound is n).

To overcome this problem, we define a new upper bound computation method based on an intermediate structure that summarizes all the cost components while maintaining part of the internal structure of what generated the cost.

Definition 5 (Cost Structure). *A* cost structure *CT is a pair $SE : CS$. Here SE is a cost expression of the form $SE = \sum_{i=1}^{n} SE_i * iv_i + e$ $(n \geq 0)$, where e is a cost expression and iv_i is a symbolic variable representing a natural number. We refer to the iv_i as* iteration variables*, to a product $SE_i * iv_i$ as* iteration component *and to SE as* structured cost expression*. CS is a (possibly empty) set of constraints of the form $\sum_{j=1}^{m} iv_j \leq e$ $(m \geq 1)$, such that all its iteration variables appear in SE. The constraints relate iteration variables with cost expressions. We use the notation $\sum iv \leq e$ when the number of iteration variables is irrelevant.*

Intuitively, a structured cost expression represents a fixed cost e plus a set of iterative components $SE_i * iv_i$, where each iterative component is executed

iv_i times and each iteration has cost SE_i. The set of constraints CS binds the values of the iteration variables iv and can express dependencies among iteration components. For instance, a constraint $iv_1 + iv_2 \leq e$ expresses that the iteration components iv_1 and iv_2 are bound by e and that the bigger iv_1 is, the smaller iv_2 must be.

We denote with IV the set of iteration variables in a cost structure. Let $val : IV \to E$ be an assignment of the iteration variables to cost expressions, a valid cost of a cost structure $CT = \sum_{i=1}^{n} SE_i * iv_i + e : CS$ is defined as $val(SE) = \sum_{i=1}^{n} val(SE_i) * val(iv_i) + e$ such that $val(CS)$ is valid.[1] A cost structure can represent multiple upper bound candidates.

Example 9. Consider a cost structure $a * iv_1 + b * iv_2 + c : \{iv_1 \leq d, iv_1 + iv_2 \leq e\}$ where a, b, c, d, and e are cost expressions. If $a > b$ and $d < e$, an upper bound is $a * d + b * nat(e - d) + c$ (The $nat()$ wrapping can be omitted). In case of $a < b$, an upper bound is $b * e + c$.

We follow a bottom up approach from S_n to S_1 and infer cost structures for cost equations, phases and chains, detailed in Secs. 4.3, 4.4, and 4.5 below. Sec. 4.2 contains a complete example. In Sec. 5, we present a technique to obtain maximal cost expressions from cost structures. They key of the procedure is to safely combine individual cost structures while detecting dependencies among them. The intermediate cost structures are correct, that is, at the end of our analysis of our example (Fig. 1) we will not only have upper bounds of *main* but also a correct upper bound of *move*.

We define the operations that form the basis or our analysis.

Definition 6 (Cost Expression Maximization). *Given a cost expression e, a cost constraint φ, and a set of variables $\overline{v}$, the operation $bd(e, \varphi, \overline{v})$ returns a set E of cost expressions that only contain variables in $\overline{v}$ and that are safe upper bounds. That is, for each $e' \in E$, we have that for all variable assignments to integers $\alpha : vars(e') \cup vars(e) \to \mathbb{Z}$ that satisfy φ: $\alpha(e') \geq \alpha(e)$. It is possible that $bd(e, \varphi, \overline{v})$ returns the empty set. In this case, no finite upper bound is known.*

For $bd(e, \varphi, \bar{v}) = \{e_1, \ldots, e_n\}$ define $\min(bd(e, \varphi, \bar{v})) = \min(e_1, \ldots, e_n)$. Note that if $bd(e, \varphi, \bar{v}) = \emptyset$, $\min(bd(e, \varphi, \bar{v})) = \infty$. Cost expression maximization can be implemented using geometrical projection over the dimensions of $\bar{v}$ in the context of the polyhedra abstract domain or as existential quantification of the variables of e and φ that do not appear in $\bar{v}$. This operation is done independently for each l in the cost expression. The results can be safely combined as linear expressions appear always inside a $nat()$ in cost expressions.

Definition 7 (Structured Cost Expression Maximization). *We define recursively the bound of a structured cost expression as $Bd(\sum_{i=1}^{n} SE_i * iv_i + e, \varphi, \bar{v}) = \sum_{i=1}^{n} Bd(SE_i, \varphi, \bar{v}) * iv_i + \min(bd(e, \varphi, \bar{v}))$.*

[1] Cost structures have some similarities to the multiple counter instrumentation described in [14]. Iteration variables can be seen as counters for individual loops or recursive components and constraints represent dependencies among these counters.

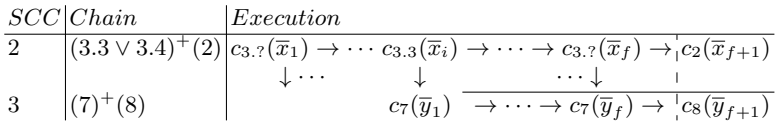


Fig. 4. Schema of executing chain $(3.3 \vee 3.4)^+(2)$

4.2 Example of upper bound computation

Fig. 4 represents the execution of chain $(3.3 \vee 3.4)^+(2)$. The execution of the phase $(3.3 \vee 3.4)^+$ consists on a series of applications of either 3.3 or 3.4. Each equation application has a call to *move*. In particular, 3.3 calls $move_{(7)^+(8)}$ and 3.4 calls $move_{(8)}$. In Fig. 4, only one call to $move_{(7)^+(8)}$ is represented. $c_n(\overline{x})$ represents an instance of cost equation n with variables $\overline{x}$.

Cost of move In order to compute the cost of the complete chain, we start by computing the cost of the innermost SCCs. In this case, the cost of *move*. The cost of one application of 8 ($c_8(\overline{y}_{f+1})$) and 7 ($c_7(\overline{y}_i)$) are 0 and 1 respectively (taken directly from the cost equations in Fig. 2). The cost of phase $(7)^+$ is the sum of the costs of all applications of c_7: $c_7(\overline{y}_1), c_7(\overline{y}_2), \cdots, c_7(\overline{y}_f)$. If c_7 is applied iv_7 times, the total cost will be $1 * iv_7$. Instead of giving a concrete value to iv_7, we collect constraints that bind its value and build a cost structure. In Sec. 3.3 we obtained the ranking function n for 7 so we have $iv_7 \leq nat(n_1)$. Moreover, the number of iterations is also bounded by $nat(n_1 - n_f)$, the difference between the initial and the final value of n in phase $(7)^+$ (see Lemma 1). Consequently, the cost structure for $(7)^+$ is $1 * iv_7 : \{iv_7 \leq n_1, iv_7 \leq n_1 - n_f\}$ (we omit the $nat()$ wrappings). If we had more ranking functions for 7, we could add extra constraints. This is important because we do not know yet which ranking function will yield the best upper bound. Additionally, we keep the cost per iteration and the number of iterations separated so we can later reason about them independently (detect dependencies). The cost of $(7)^+(8)$ is the cost of $(7)^+$ plus the cost of (8) but expressed according to the initial variables $\overline{y}_1$. We add the cost structures and maximize them (Bd) using the corresponding invariants. We obtain $1 * iv_7 : \{iv_7 \leq n_1, iv_7 \leq n_1 - no_1\}$ (because $n_f > n_{f+1} = no_{f+1} = no_1$).

Cost of one application of $3.3, 3.4$ *and* 2 The cost of (2) is 0. The cost of one application of 3.4 is the cost of a call to $move_{(8)}$, that is, 0. Conversely, the cost of one application of 3.3 is the cost of one call to $move_{(7)^+(8)}$. We want the cost of $c_{3.3}(\overline{x}_i)$ expressed in terms of the entry variables $\overline{x}_i$ and the variables of the corresponding recursive call $\overline{x}_{i+1}$. We maximize the cost structure of $move_{(7)^+(8)}$ using the cost constraints of 3.3 ($\varphi_{3.3}$). This results in the cost structure $1 * iv_7 : \{iv_7 \leq n_i, iv_7 \leq n_i - n_{i+1}\}$ (the output *no* is n_{i+1} in the recursive call).

Cost of phase $(3.3 \vee 3.4)^+$ The cost of phase $(3.3 \vee 3.4)^+$ is the sum of the cost of all applications of $c_{3.3}$ and $c_{3.4}$: $c_{3.?}(\overline{x}_1), c_{3.?}(\overline{x}_2), \cdots, c_{3.?}(\overline{x}_f)$. We group the

summands originating from 3.3 and from 3.4 and assume that $c_{3.3}$ and $c_{3.4}$ are applied $iv_{3.3}$ and $iv_{3.4}$ times respectively. The sum of all applications of $c_{3.4}$ is $0 * iv_{3.4} = 0$. However, the cost of each $c_{3.3}(\overline{x}_i)$ might be different (depends on $\overline{x}_i$) so we cannot simply multiply. Using the invariant $\varphi_{(3.3 \vee 3.4)^*}$ and $\varphi_{3.3}$ we know that $n_1 \geq n_i \wedge n_i > n_{i+1} \wedge n_{i+1} \geq 0$. Maximizing each of these constraints yields $iv_7 \leq n_1$ and we obtain a cost structure $1 * iv_7 : \{iv_7 \leq n_1\}$ that is greater or equal than all $1 * iv_7 : \{iv_7 \leq n_i, iv_7 \leq n_i - n_{i+1}\}$ (because $n_1 \geq n_i$). Therefore, a valid (but imprecise) cost of $(3.3 \vee 3.4)^+$ is $(1 * iv_7) * iv_{3.3} : \{iv_7 \leq n_1, iv_{3.3} \leq n_1, iv_{3.3} \leq n_1 - n_f\}$ (n is a ranking function of 3.3). If we solve the cost structure, we will obtain the upper bound n^2.

Inductive constraint compression Because we kept the different components of the cost separated, we can easily obtain a more precise cost structure Each call to *move* starts where the last one left it and all of them together can iterate at most n times. This is reflected by the constraint $iv_7 \leq n_i - n_{i+1}$. We can compress all the iterations $(n_1 - n_2) + (n_2 - n_3) + \cdots + (n_{f-1} - n_f) \leq n_1 - n_f$, pull out the iteration component $1 * iv_7$ and obtain a more precise cost structure $(1 * iv_7) + (0 * iv_{3.3}) : \{iv_7 \leq n_1 - n_f, iv_{3.3} \leq n_1, iv_{3.3} \leq n_1 - n_f\}$. Then, we can eliminate $(0 * iv_{3.3})$ arriving at $(1 * iv_7) : \{iv_7 \leq n_1 - n_f\}$ which will result in an upper bound n.

4.3 Cost Structure of an Equation Application

Consider a cost equation $c = \langle C(\bar{x}) = \sum_{i=1}^{n} D_i(\bar{y}_i) + e + C(\bar{x}'), \varphi \rangle$, where $C(\bar{x}')$ is a recursive call. We want to obtain a cost structure $SE_c : CS_c$ that approximates the cost of $\sum_{i=1}^{n} D_i(\bar{y}_i) + e$ and we want such a cost structure to be expressed in terms of $\bar{x}$ and $\bar{x}'$.

Example 10. Consider cost equation 3.3 from Fig. 3 which is part of SCC S_2:

$$while(n, m, fwd) = move_{(7)^+(8)}(n'', m'', fwd'', no) + while(n', m', fwd')$$

Assume φ contains $n'' = n \wedge n' = no$. The cost of one application of 3.3 is the cost of $move_{(7)^+(8)}(n, m, fwd, no)$ expressed in terms of n, m, fwd and n', m', fwd'. Let the cost of $move_{(7)^+(8)}$ be $1 * iv_7 : \{iv_7 \leq n'', iv_7 \leq n'' - no\}$, then we obtain an upper bound by maximizing the structured cost expression and the constraints in terms of the variables n, m, fwd and n', m', fwd'. The obtained cost structure is $1 * iv_7 : \{iv_7 \leq n, iv_7 \leq n - n'\}$.

Let $SE_i : CS_i$ be the cost structure of the chain D_i, then the structured cost expression can be computed as $SE_c = \sum_{i=1}^{n} Bd(SE_i, \varphi, \bar{x}) + min(bd(e, \varphi, \bar{x}))$. By substituting each call $D_i(\bar{y}_i)$ by its structured cost expression and maximizing with respect to $\bar{x}$, we obtain a valid structured cost expression in terms of the entry variables.

A set of valid constraints CS_c is obtained simply as the union of all sets CS_i expressed in terms of the entry and recursive call variables ($\bar{x}$ and $\bar{x}'$): $CS_c \supseteq \{\sum iv \leq e' | \sum iv \leq e \in CS_i, e' \in bd(e, \varphi, \bar{x}\bar{x}')\}$. Should the cost equation not have a recursive call, all the maximizations will be performed only with respect to the entry variables $\bar{x}$.

Constraint Compression In order to obtain tighter bounds, one can try to detect dependencies among the constraints when they have a linear cost expression. Let $\sum iv_i \leq nat(l_i) \in CS_i$ and $\sum iv_j \leq nat(l_j) \in CS_j$, $j \neq i$. Now assume there exist $l_{new} \in bd(l_i + l_j, \varphi, \bar{x}\bar{x'})$, $l'_i \in bd(l_i, \varphi, \bar{x}\bar{x'})$, and $l'_j \in bd(l_j, \varphi, \bar{x}\bar{x'})$ such that $\varphi \Rightarrow (l_{new} \leq (l'_i + l'_j) \wedge l_{new} \geq l_i \wedge l_{new} \geq l_j)$. $nat(l_{new})$ might bind $nat(l_i)$ and $nat(l_j)$ tighter than $nat(l'_i)$ and $nat(l'_j)$. Then we can add $\sum iv_i + \sum iv_j \leq nat(l_{new})$ to the new set of constraints CS_c.

Example 11. Suppose the cost equation from the previous example had two consecutive calls to *move*: $while(n, m, fwd) = move_{(7)^+(8)}(n_1, m_1, fwd_1, no_1) + move_{(7)^+(8)}(n_2, m_2, fwd_2, no_2) + while(n', m', fwd')$ with $\{n_1 = n \wedge no_1 = n_2 \wedge no_2 = n'\} \subseteq \varphi$. The resulting cost structure would be $1 * iv_{7.1} + 1 * iv_{7.2} * 2 : \{iv_{7.1} \leq n, iv_{7.1} \leq n - n', iv_{7.2} \leq n, iv_{7.2} \leq n - n'\}$ ($iv_{7.1}$ and $iv_{7.2}$ correspond to the iterations of the two instances of phase $(7)^+$). However, we could compress $iv_{7.1} \leq n_1 - no_1$ and $iv_{7.2} \leq n_2 - no_2$ (from Ex. 10) into $iv_{7.1} + iv_{7.2} \leq n - n'$ and add it to the final set of constraints. This set represents a tighter bound and captures the dependency between the first and the second call.

4.4 Cost Structure of a Phase

Refined phases have the form of a single equation (c) or an iterative phase $(c_1 \vee c_2 \vee \ldots \vee c_n)^+$. The cost of (c) is simply the cost of c. The cost of an iterative phase is the sum of the costs of all applications of each c_i (see Sec. 4.2). Let $CT_i = SE_i : CS_i$ be the cost of one application of c_i, we group the summands according to each c_i and assign a new iteration variable iv_i that represents the number of times such a cost equation is applied. The total cost of the phase is $\sum_{i=1}^{n}(\sum_{j=1}^{iv_i} SE_i(x_j))$ where $SE_i(x_j)$ is an instance of SE_i with the variables corresponding to the j-th application of c_i.

For each c_i in the phase $(c_1 \vee c_2 \vee \ldots \vee c_n)^+$ we obtain a structured cost expression $Bd(SE_i, \varphi_{ph^*}, \bar{x}_1)$ where φ_{ph^*} is an auxiliary invariant that relates $\bar{x}_1$ (the variables at the beginning of the phase) to any $\bar{x}_j$ as defined in Sec. 3.2. That structured cost expression is valid for any application of c_i during the phase. This allows us to transform each sum $\sum_{j=1}^{iv_i} SE_i(\bar{x}_j)$ into a product $iv_i * Bd(SE_i, \varphi_{ph^*}, \bar{x}_1)$. Similarly, we maximize the cost expressions in the constraints. A set of valid constraints is $CS_{ph} = \bigcup_{i=1}^{n}(\{\sum iv_i \leq e'_i | \sum iv_i \leq e_i \in CS_i, e'_i \in bd(e_i, \varphi_{ph^*} \wedge \varphi_{c_i}, \bar{x}_1\}) \cup CS_{new}$, where CS_{new} is a new set of constraints that bounds the new iteration variables $(iv_1, iv_2, \cdots, iv_n)$. The maximization of the constraints is equivalent to the maximization of the iteration variables inside SE_i (a proof can be found in [11]).

Bounding the iterations of a phase To generate the constraints in CS_{new}, we use the ranking functions and their dependencies obtained when proving termination (see Sec. 3.3).

Example 12. Consider a phase formed by the following cost equations expressed in compact form (we assume that all have the condition $a, b, c \geq 0$):

$1 : p(a,b,c) = p(a-1,b,c) \,|\, 2 : p(a,b,c) = p(a+2,b-1,c) \,|\, 3 : p(a,b,c) = p(a,c,c-1)$
(3) has a ranking function c with no dependencies. We can add $iv_3 \leq c$ to the constraints. (2) has b as a ranking function but it depends on (3). Every time (3) is executed, b is "restarted". Fortunately, the value assigned to b has a maximum (the initial c). Therefore, we can add the constraint $iv_2 \leq b + c * c$. Finally, (1) has a as a ranking function that depends on (2). a is incremented by 2 in every execution of (2) whose number of iterations is at most $b + c * c$. We add the constraint $iv_1 \leq a + 2 * (b + c * c)$.

More formally, we have a set RF_i for each c_i in a phase. Each $f \in RF_i$ has a (possibly empty) dependency set to other c_j. Given a ranking function f that occurs in all sets $RF_{i_1}, \ldots, RF_{i_m}$ for a maximal m, $i_k \in 1..n$. If f has no dependencies, then $nat(f)$ expressed in terms of $\bar{x}_1$ is an upper bound on the number of iterations of $c_{i_1}, \ldots, c_{i_m}$ and we add $\sum_{k=1}^{m} iv_{i_k} \leq nat(f)$ to CS_{new}.

If f depends on $c_{j_1}, \ldots, c_{j_l}$ ($j_i \in 1..n$) and $ub_{j_1}, \ldots, ub_{j_l}$ are upper bounds on the number of iterations of $c_{j_1}, \ldots, c_{j_l}$, then we distinguish two types of dependencies: (1) if c_{j_i} increases f by a constant t_{j_i} then each execution of c_{j_i} can imply t_{j_i} extra iterations. We add $ub_{j_i} * t_{j_i}$ to f; (2) otherwise, if f can be "restarted" in every execution of c_{j_i}, then $R^f_{j_i} \in bd(f(\bar{x}_3), \varphi_{ph^*}(\bar{x}_1\bar{x}_2) \wedge \varphi_{c_{j_i}}(\bar{x}_2\bar{x}_3), \bar{x}_1)$ represents the maximum value that f can take in c_{j_i} (if it exists) and we add $ub_{j_i} * nat(R^f_{j_i})$. Taken together, we can add $\sum_{k=1}^{m} iv_{i_k} \leq nat(f) + \sum_{i=1}^{p} ub_{j_i} * t_{j_i} + \sum_{i=p}^{l} ub_{j_i} * nat(R^f_{j_i})$ to CS_{new} where $c_{j_1}, c_{j_2} \cdots c_{i_p}$ are the dependencies of type (1) and $c_{i_p}, c_{i_{p+1}} \cdots c_{i_l}$ the ones of type (2).

On top of this, we add constraints that depend on the value of the variables after the phase (see the cost of $(7)^+$ Sec.4.2). This will allow us to perform constraint compression afterwards.

Lemma 1. *Given a sequence of r calls $c_{i_1}(\bar{x}_1) \cdot c_{i_2}(\bar{x}_2) \cdots c_{i_r}(\bar{x}_r) \cdot c'(\bar{x}_{r+1})$, during which c_i occurred p times and $f \in RF_i$, and for all $\langle c_{i_j}(\bar{x}_j) = \cdots + c_{i_{j+1}}(\bar{x}_{j+1}), \varphi\rangle$, $\varphi \Rightarrow (f(\bar{x}_j) - f(\bar{x}_{j+1}) \geq 0)$. We have that $f(\bar{x}_1) - f(\bar{x}_{r+1}) \geq p$.*

If f is a ranking function in $RF_{i_1}, \ldots, RF_{i_m}$ as above, if f has no dependencies, we can use Lemma 1 (a proof can be found in [11]) to add $\sum_{k=1}^{m} iv_{i_k} \leq nat(f(\bar{x}_1) - f(\bar{x}_f))$ to CS_{new} where $\bar{x}_f$ are the variables at the end of the phase.

Inductive constraint compression We generalize the *constraint compression* presented in Sec. 4.3. Instead of compressing two constraints, we compress an arbitrary number of them inductively. This is the mechanism used to obtain a linear bound for the chain $(3.3 \vee 3.4)^+$ at the end of Sec. 4.2.

When a constraint is compressed, its iteration variables should be removed from constraints that cannot be compressed. Removing an iteration variable from a constraint is always safe but can introduce imprecision.

Given a cost expression e_i that we want to compress to $\sum iv \leq e_i$, we start with a copy e'_i of e_1 as our candidate. First, prove the base case $\varphi_i \Rightarrow e'_i \geq e_i$ (which is trivial given that e_i and e'_i are equal). Then prove the induction step $\varphi_{ph}(\bar{x}_1\bar{x}_2) \wedge \varphi_{ph^*}(\bar{x}_2\bar{x}_3) \wedge \varphi_i(\bar{x}_3\bar{x}_4) \Rightarrow e'_i(\bar{x}_1\bar{x}_4) \geq e'_i(\bar{x}_1\bar{x}_2) + e_i(\bar{x}_3\bar{x}_4)$. Assuming e'_i

is valid for a number of iterations (represented as $\varphi_{ph}(\bar{x}_1\bar{x}_2)$), this shows that it is valid for one more iteration ($\varphi_i(\bar{x}_3\bar{x}_4)$) even if there are interleavings with other c_j ($\varphi_{ph^*}(\bar{x}_2\bar{x}_3)$). Once we proved that, we can add the constraint $\sum iv' \leq e'_i$ and pull the corresponding iteration components out of the corresponding product (a proof can be found in [11]).

If we can prove the stronger inequality $e'_i(\bar{x}_1\bar{x}_4) \geq e'_i(\bar{x}_1\bar{x}_2) + e_i(\bar{x}_3\bar{x}_4) + 1$, then we know that e'_i also decreases with the iterations of c_i. In this case we derive a new constraint $\sum iv' + iv_i \leq e'_i$. We can generalize this procedure to compress constraints that originate from different equations. This is demonstrated by the following example.

Example 13. Consider the phase $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$. Both 3.1 and 4.1 have a call to $move_{(5)^+(6)}$ and their cost structures are $iv_{5.1} * 2 : \{iv_{5.1} \leq n' - n, iv_{5.1} \leq m - n\}$ and $iv_{5.2} * 2 : \{iv_{5.2} \leq n' - n, iv_{5.2} \leq m - n\}$. We can compress both iteration variables obtaining $iv_{5.1} * 2 + iv_{5.2} * 2 : \{iv_{5.1} + iv_{5.2} \leq n' - n\}$ (3.2 and 4.2 have zero cost) that when maximized will give us $iv_{5.1} * 2 + iv_{5.2} * 2 : \{iv_{5.1} + iv_{5.2} \leq m - n\}$ which represents the upper bound $2(m - n)$.

4.5 Cost Structure of a Chain

Given a chain $ch = ph_1 \cdots ph_n$ whose phases have cost structures $CT_1, \ldots CT_n$, we want to obtain a cost structure $CT_{ch} = SE_{ch} : CS_{ch}$ for the total cost of the chain. This is analogous to computing the cost structure of an equation in Sec. 4.3. One constructs a cost constraint φ_{ch} relating all variables of the calls to the entry variables and to each other: $\varphi_{ch} = \varphi_{ph_1}(x_1x_2) \wedge \varphi_{ph_2}(x_2x_3) \wedge \cdots \wedge \varphi_{ph_n}(x_n)$. This cost constraint can be enriched with the invariants of the chain.

The structured cost expression is $SE_{ch} = \sum_{i=1}^{n} Bd(SE_i, \varphi_{ch}, \bar{x})$ and the constraints are $CS_c \supseteq \{\sum iv \leq e' | \sum iv_i \leq e \in CS_i, e' \in bd(e, \varphi_{ch}, \bar{x})\}$. Again, we can apply *constraint compression* to combine constraints from different phases.

Example 14. The cost of patterns (2) and (3) in Ex. 7 derive directly from the cost of their phases (see Sec. 4.2 and Ex. 13). We examine the cost of pattern (1), that is, $(3.3 \vee 3.4)^+(4.3)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$. Considering that variables are subscripted with 1, 2 and 3 for their value before the first, second and third phase, the cost structures of the phases are: $1 * iv_{7.1} : \{iv_{7.1} \leq n_1 - n_2\}$, $1 * iv_{7.2} : \{iv_{7.2} \leq n_2 - n_3\}$ and $iv_{5.1} * 2 + iv_{5.2} * 2 : \{iv_{5.1} + iv_{5.2} \leq n_4 - n_3\}$. The joint invariants guarantee that $n_3 \geq 0 \wedge n_4 \leq m$. We can compress the constraints $iv_{7.1} \leq n_1 - n_2$ and $iv_{7.2} \leq n_2 - n_3$ and maximize with respect to the initial variables obtaining $1 * iv_{7.1} + 1 * iv_{7.2} + 2 * iv_{5.1} + 2 * iv_{5.2} : \{iv_{7.1} + iv_{7.2} \leq n_1, iv_{5.1} + iv_{5.2} \leq m_1\}$. Such a cost structure represents the bound $n + 2m$ as expected.

5 Solving Cost Structures

Solving a cost structure $SE : CS$ means to look for a maximizing assignment val_{max} from iteration variables to cost expressions (without iteration variables) such that $CS \Rightarrow val_{max}(SE) \geq SE$ is valid. Even though iteration variables

range over natural numbers, we consider a relaxation of the problem where iteration variables can take any non-negative real number. The maximization of $val_{max}(SE)$ represents the cost structure SE where each iv has been substituted by $val_{max}(iv)$ and $val_{max}(SE)$ is an upper bound of the cost structure $SE : CS$.

Let $SE = \sum_{i=1}^{n} SE_i * iv_i + e$, The maximization of each SE_i can be performed independently, because its iteration variables depend neither on other iteration variables of SE_j for $j \neq i$ nor on any iv_i. Let e_i be the maximization of SE_i, then we obtain $\sum_{i=1}^{n} e_i * iv_i + e$ as well as a set of constraints over the iv_i. As the e_i's can be symbolic expressions, not necessarily comparable to each other, we need a procedure to find an upper bound independently of the e_i.

We group iteration components (Def. 5) based on dependencies. Two iteration components depend on each other if their iteration variables appear together in a constraint. An iteration group IG is a partial cost structure $\sum_{i=1}^{m} e_{j_i} * iv_{j_i} : CS$ ($1 \leq j_i \leq n$ for $i \in 1..m$) where its iteration components depend on each other.

A constraint $\sum_{i=1}^{m} iv_{j_i} \leq e$ is *active* for assignment val iff $\sum_{i=1}^{m} val(iv_{j_i}) = e$. Let $C = \sum_{i=1}^{m} iv_{j_i} \leq e$, $C' = \sum_{i=1}^{m+k} iv_{j_i} \leq e'$ be constraints such that $C \subseteq C'$ and val any assignment: (i) If C is active for val, then $C = e$ and we substitute $\sum_{i=m+1}^{m+k} iv_{j_i} \leq nat(e' - e)$ for C' making the two constraints independent; (ii) If C is not active, we ignore C and consider the rest of the constraints.

Consider an IG $SE : CS$ that we want to maximize. For each $C, C' \in CS$ with $C \subseteq C'$, we use the observation in the previous paragraph to derive simplified constraints CS_1, CS_2. We solve both constraints and obtain val_1, val_2. The maximum cost of IG is $\min(val_1(SE), val_2(SE))$. Constraints with only one iv can always be reduced. We repeat the procedure until the constraints cannot be further simplified. The constraints can now be grouped into irreducible IGs. A trivial IG is one with a single iv constraint $iv \leq e$ whose maximal assignment is $val(iv) = e$. All constraints in an irreducible, non-trivial IG have at least two iteration variables.

Example 15. Consider the following cost structure $iv_1 * 1 + iv_2 * (b) + iv_3 * (iv_4 * 2) : \{iv_1 + iv_2 + iv_3 \leq a + b, iv_1 + iv_2 \leq c, iv_4 \leq d\}$. First, we maximize the internal iteration component $iv_4 * 2$ which contains a trivial IG $iv_4 \leq d$. The result is $iv_1 * 1 + iv_2 * (b) + iv_3 * (2d) : \{iv_1 + iv_2 + iv_3 \leq a + b, iv_1 + iv_2 \leq c\}$. This cost structure forms a single IG with two constraints one contained in the other. (1) We assume $iv_1 + iv_2 \leq c$ is active. Then we have $\{iv_3 \leq nat(a + b - c), iv_1 + iv_2 \leq c\}$ which contains two irreducible IG. The first one is $iv_3 = nat(a + b - c)$ and the second one has two possibilities $iv_1 = c, iv_2 = 0$ or $iv_1 = 0, iv_2 = c$ (Thm. 2 below). The result is then $nat(a+b-c)+max(b*c, 2d*c)$. (2) If $iv_1 + iv_2 \leq c$ is not active, we have only $iv_1 + iv_2 + iv_3 \leq a + b$ which yields $max(a+b, b*(a+b), 2d*(a+b))$. The cost is $min(nat(a + b - c) + max(b * c, 2d * c), max(a + b, b * (a + b), 2d * (a + b)))$.

We could have dropped the second constraint from the beginning and obtain a less precise bound $max(a + b, b * (a + b), 2d * (a + b))$. We can even split the constraint $iv_1 + iv_2 + iv_3 \leq a + b$ into $iv_1 \leq a + b, iv_2 \leq a + b$ and $iv_3 \leq a + b$ and obtain $(1 + b + 2d) * (a + b)$. That way we can balance precision and performance.

Definition 8 (IG dependency graph). *Let $IG = SE : CS$. Its dependency graph $G(IG)$ is defined as follows: for each $C \in CS$ G has a node C. For each*

$C \cap C'$ such that $C, C' \in CS$ and $C \cap C' \neq \emptyset$ G has a node $d(C \cap C')$, and edges from C to $d(C \cap C')$ and from $d(C \cap C')$ to C'.

Example 16. Given the IG $\{iv_1 + iv_2 \leq a, iv_2 + iv_3 \leq b, iv_2 + iv_4 \leq c\}$, its dependency graph contains the nodes $n_1 =$ "$iv_1 + iv_2 \leq a$", $n_2 =$ "$iv_2 + iv_3 \leq b$", $n_3 =$ "$iv_2 + iv_4 \leq c$" and $n_4 =$ "$d(iv_2)$". The edges are $(n_1, n_4), (n_2, n_4), (n_3, n_4)$.

Theorem 2. *Given an irreducible, non-trivial IG. If $G(IG)$ is acyclic there exists a maximizing assignment val_{max} such that there is an active constraint with only one non-zero iteration variable.*

If $G(IG)$ is acyclic, we apply Thm. 2 to solve IG incrementally. Let $C_i = \sum_{j=1}^{r} iv_{i_j} \leq e \in CS$: we obtain a partial assignment val_{ik} such that $val_{ik}(iv_k) = e$ for some $iv_k \in C_i$ and all other iteration variables in C_i being assigned 0. We update CS with val_{ik} and obtain a constraint system with less iteration variables and constraints whose graph is still acyclic, and so on. Once no iteration variable is left, we end up with a set of assignments $MaxVal$. The maximum cost of $IG = SE : CS$ is $max_{val \in MaxVal} val(SE)$.

Example 17. We obtain one of the assignments in $MaxVal$ for the IG of Ex. 16. We take the constraint $iv_1 + iv_2 \leq a$ and assign $iv_1 = a$ and $iv_2 = 0$. The resulting constraints are $iv_3 \leq b$ and $iv_4 \leq c$ that are trivially solved. The resulting assignment is $iv_1 = a, iv_2 = 0, iv_3 = b$ and $iv_4 = c$.

The requirement of $G(IG)$ being acyclic can be relaxed. A discussion and the proof of Thm. 2 is in [11]. One can always obtain an acyclic IG by dropping constraints or by removing iteration variables from a given constraint. Such transformations are safe since they only relax the conditions imposed on the iteration variables. In practice, we perform a pre-selection of the constraints to be considered based on heuristics to improve performance.

6 Related Work and Experiments

This work builds upon the formalism developed in the COSTA group [1, 2, 4, 5], however, the are important differences in how upper bounds are inferred. In [1], upper bounds are computed independently for each SCC and then combined without taking dependencies into account. The precision of that approach is improved in [2] for certain kinds of loops. The paper [5] presents a general approach for obtaining amortized upper bounds that, although powerful, does not scale well. In [4] SCCs are decomposed into sparse cost equations systems. Then it is possible to use the ideas of [5] to solve the sparse cost equations precisely.

In our work, we also decompose programs, but driven by possible sequences of cost applications. This technique, known as control-flow-refinement, has been applied to the resource analysis of imperative programs in [13, 9]. In addition, our refinement technique can deal with programs with linear recursion (non neccessarily tail recursive) and multiple procedures. In our analysis we do not refine the whole program at once. Instead, we refine each SCC and then propagate

the changes. Our technique allows to leave parts of the program unrefined to increase performance. Paper [15] uses disjunctive invariants to summarize inner loops instead of control-flow-refinement. This technique can also deal with some kinds of non-terminating programs. However, it can only bound the number of visits to a single location in a single procedure. In contrast, our tool can count the number of visits to several locations in multiple procedures derived from cost annotations. The tool Loopus [18] uses disjunctive invariants, collects the inner paths of each loop and also uses contextualization which is a form of control-flow refinement. Both [15, 18] obtain ranking functions based on given patterns and combine them using proof rules. Instead, we infer linear ranking functions using linear programming [16, 6] and combine them to form lexicographic ranking functions (see Sec. 4.4).

SPEED [14] makes use of multiple counters to bound and detect dependencies of different loops. SPEED computes cost summaries for the (non-recursive) procedure calls. Therefore, it cannot detect dependencies among different procedure calls. KoAT [8] adopts an iterative approach, where size analysis and complexity analysis are interleaved and improve each other. That paper also extends transitions systems to deal with inter-procedural and recursive programs. Very recently, a new version of Loopus has been released [17]. They use a simple abstraction and achieve very high performance and great effectiveness. They can also obtain amortized cost for complex nested loops. However, their analysis is limited to imperative programs and cannot deal with recursion.

	1	$log\ n$	n	$n\ log\, n$	n^2	n^3	$> n^3$	No res.
CoFloCo	115	0	141	0	52	2	3	318
KoAT	117	0	120	0	51	0	4	339
PUBS	90	2	85	5	37	3	3	406
Loopus[3]	128	0	140	0	73	11	4	275
CoFloCo	1	0	16	0	14	7	0	1
PUBS	1	2	13	3	12	6	0	2
Loopus[3]	2	0	11	0	7	4	0	15

For our experimental evaluation we took the problem set used by KoAT's evaluation[2] [8], except those with multiple recursion (670 problems). We executed each problem with PUBS [1], KoAT, and our tool CoFloCo (SPEED and the first version of Loopus [18] are not publicly available). The problems are taken from the literature on resource analysis [3, 13–15, 18] and include most of the problems used in the evaluation of [7] (631 problems in the first part of the table) and the ones of the evaluation of PUBS [1] (39 problems in the second part).

The problems of the first part were automatically translated from KoAT's input format to cost equations. That includes performing loop extraction (and generating invariants for PUBS). No slicing took place so the input cost equations might have many more variables than needed. For the second set we used the original cost equations for PUBS and CoFloCo. We decided not to include these problems for KoAT as the translation generated in [8] is not sound (we found several problems where KoAT yields an incorrect upper bound). We summarize the number of problems solved by each tool in different complexity categories.

[2] `http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity/`

Each problem was run with a time-out of 60 secs. The same set of problems[3] has been used to evaluate the new version of Loopus [17]. We include the results of their evaluation[4] in a shaded row to emphasize that we did not run the experiments ourselves.

CoFloCo obtains a bound asymptotically better than KoAt in 60 problems and better than PUBS in 109 problems. Conversely, KoAt obtains a better bound than CoFloCo in 23 problems and PUBS is better than CoFloCo in 11 problems. CoFloCo obtains better results than Loopus in 48 of the problems analyzed by both. Loopus obtains better results than CoFloCo in 93 problems. However, in 51 of these problems, Loopus reports an upper bound as a function of `call_to_nondet_line_X` where `X` is a line number. It seems that Loopus assumes a specific symbolic value whenever a non-deterministic assignment is executed whereas CoFloCo does not make such an assumption and fails to provide a bound. The complete experimental data and the implementation are available.[5]

At this time, CoFloCo is just prototype and can be greatly improved. It fails on 27 problems because of irreducible loops. Irreducible loops can be transformed and the approach could be extended to handle other domains including non-linear constraints, logarithmic bounds, etc. The invariants could also be improved with the termination information of Sec.3.3 following the ideas of [8]. CoFloCo had 94 time-outs. Most occurred with problems with many variables where slicing could be applied. In some occasions, the control-flow-refinement of cost equations can generate exponentially many chains. However, these chains have many fragments in common and part of the invariant and upper bound computation can be reused. Moreover, some SCCs can be left unrefined to achieve a compromise between performance and precision.

We presented a control-flow-refinement algorithm that can be applied to linear recursive programs (other approaches do not support recursion). The algorithm distinguishes terminating and non-terminating executions explicitly which allows obtaining better invariants for the terminating executions. This also allows to have intermediate cost expressions depending on the output variables (see the cost of $(7)^+(8)$) and thus obtain amortized cost bounds. We obtain an upper bound for each execution pattern (chain), which often provides more precise information than a generic upper bound for any possible execution. The upper bounds are also precise because cost structures allow us to maintain several upper bound candidates, detect dependencies among different parts of the code (using *constraint compression*) and obtain complex upper bound expressions.

Acknowledgements Research partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services. We thank the anonymous reviewers for their careful reading which resulted in numerous improvements. We thank S. Genaim for valuable discussions and help with the experiments.

[3] 18 problems included here were left out of the evaluation of Loopus.

[4] `http://forsyte.at/static/people/sinn/loopus/CAV14/`

[5] `www.se.tu-darmstadt.de/se/group-members/antonio-flores-montoya/cofloco`

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. of Automated Reasoning*, 46(2):161–203, Feb. 2011.
2. E. Albert, S. Genaim, and A. N. Masud. More precise yet widely applicable cost analysis. In *VMCAI, Austin, TX*, volume 6538 of *LNCS*, pages 38–53. Springer, 2011.
3. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.
4. D. E. Alonso-Blas, P. Arenas, and S. Genaim. Precise cost analysis via local reasoning. In *ATVA*, LNCS. Springer, oct 2013.
5. D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *SAS*, volume 7460 of *LNCS*, pages 405–421. Springer, Sept. 2012.
6. R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215(0):47 – 67, 2012.
7. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 413–429. Springer, 2013.
8. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, 2014.
9. H. Chen, S. Mukhopadhyay, and Z. Lu. Control flow refinement and symbolic computation of average case bound. In *Automated Technology for Verification and Analysis*, volume 8172 of *LNCS*, pages 334–348. Springer, 2013.
10. B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 47–61. Springer, 2013.
11. A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. Technical report, TU Darmstadt, 2014. `https://www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Page_Content/Group_Members/Antonio_Flores-Montoya/APLAS14_techReport.pdf`.
12. B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Computer Aided Verification*, volume 5123 of *LNCS*, pages 370–384. Springer, 2008.
13. S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.
14. S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, New York, NY, USA, 2009. ACM.
15. S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI'10*, pages 292–304, New York, NY, USA, 2010. ACM.
16. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, volume 2937 of *LNCS*, pages 239–251, 2004.
17. M. Sinn, F. Zuleger, and H. Veith. A simple and scalable approach to bound analysis and amortized complexity analysis. In *CAV*, volume 8559 of *LNCS*, pages 743–759. Springer, 2014.
18. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In E. Yahav, editor, *Static Analysis*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

Appendix K

Article *SACO: Static Analyzer for Concurrent Objects*, [3]

SACO: Static Analyzer for Concurrent Objects

E. Albert[1], P. Arenas[1], A. Flores-Montoya[2], S. Genaim[1],
M. Gómez-Zamalloa[1], E. Martin-Martin[1], G. Puebla[3], and G. Román-Díez[3]

[1] Complutense University of Madrid (UCM), Spain
[2] Technische Universität Darmstadt (TUD), Germany
[3] Technical University of Madrid (UPM), Spain

Abstract. We present the main concepts, usage and implementation of SACO, a static analyzer for concurrent objects. Interestingly, SACO is able to infer both *liveness* (namely termination and resource boundedness) and *safety* properties (namely deadlock freedom) of programs based on concurrent objects. The system integrates auxiliary analyses such as *points-to* and *may-happen-in-parallel*, which are essential for increasing the accuracy of the aforementioned more complex properties. SACO provides accurate information about the dependencies which may introduce deadlocks, loops whose termination is not guaranteed, and upper bounds on the resource consumption of methods.

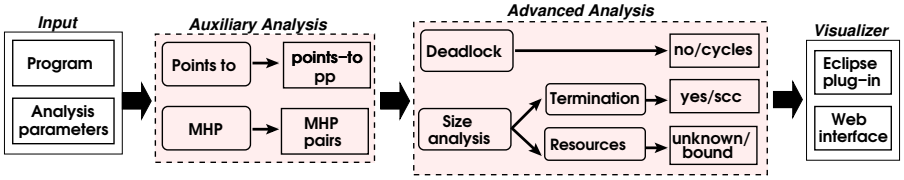
1 Introduction

With the trend of parallel systems, and the emergence of multi-core computing, the construction of tools that help analyzing and verifying the behaviour of concurrent programs has become fundamental. Concurrent programs contain several processes that work together to perform a task and communicate with each other. Communication can be programmed using shared variables or message passing. When shared variables are used, one process writes into a variable that is read by another; when message passing is used, one process sends a message that is received by another. Shared memory communication is typically implemented using low-level concurrency and synchronization primitives These programs are in general more difficult to write, debug and analyze, while its main advantage is efficiency. The message passing model uses higher-level concurrency constructs that help in producing concurrent applications in a less error-prone way and also more modularly. Message passing is the essence of actors [1], the concurrency model used in *concurrent objects* [9], in Erlang, and in Scala.

This paper presents the SACO system, a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects. Essentially, each concurrent object is a monitor and allows at most one *active* task to execute within the object. Scheduling among the tasks of an object is cooperative, or non-preemptive, such that the active task has to release the object lock explicitly (using the **await** instruction). Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and start executing. When the result of a call is required by the caller to continue executing, the caller and the

callee methods can be synchronized by means of *future variables*, which act as proxies for results initially unknown, as their computations are still incomplete.

The figure below overviews the main components of SACO, whose distinguishing feature is that it infers both *liveness* and *safety* properties.



SACO receives as input a program and a selection of the analysis parameters. Then it performs two auxiliary analyses: points-to and may-happen-in-parallel (MHP), which are used for inferring the more complex properties in the next phase. As regards to liveness, we infer termination as well as resource boundedness, i.e., find upper bounds on the resource consumption of methods. Both analyses require the inference of *size relations*, which are gathered in a previous step. Regarding *safety*, we infer deadlock freedom, i.e., there is no state in which a non-empty set of tasks cannot progress because all tasks are waiting for the termination of other tasks in the set, or otherwise we show the tasks involved in a potential deadlock set. Finally, SACO can be used from a command line interface, a web interface, and an Eclipse plugin. It can be downloaded and/or used online from its website http://costa.ls.fi.upm.es/saco.

2 Auxiliary Analyses

We describe the auxiliary analyses used in SACO by means of the example below:

```
1 class PrettyPrinter{
2   void showIncome(Int n){...}
3   void showCoin(){...}
4 }//end class
5 class VendingMachine{
6 Int coins;
7 PrettyPrinter out;
8   void insertCoins(Int n){
9     Fut⟨void⟩ f;
10    while (n>0){
11      n=n−1;
12      f=this ! insertCoin();
13      await f?; }
14 }
15   void insertCoin(){
16    coins=coins+1;
17 }
18   Int retrieveCoins(){
19    Fut⟨void⟩ f;
20    Int total=0;
21    while (coins>0){
22      coins=coins−1;
23      f=out ! showCoin();
24      await f?;
25      total=total+1; }
26    return total;
27   }
28 }//end class
29 //main method
30 main(Int n){
31   PrettyPrinter p;
32   VendingMachine v;
33   Fut⟨Int⟩ f;
34   p=new PrettyPrinter();
35   v=new VendingMachine(0,p);
36   v ! insertCoins(n);
37   f=v ! retrieveCoins();
38   await f?;
39   Int total=f.get;
40   p!showIncome(total);
41 }
```

We have a class PrettyPrinter to display some information and a class VendingMachine with methods to insert a number of coins and to retrieve all coins. The main method is executing on the object This, which is the initial object, and receives as parameter the number of coins to be inserted. Besides This, two other concurrent objects are created at Line 34 (L34 for short) and L35. Objects can be

seen as buffers in which tasks are posted and that execute in parallel. In particular, two tasks are posted at L36 and L37 on object v. insertCoins executes asynchronously on v. However, the **await** at L38 synchronizes the execution of This with the completion of the task retrieveCoins in v by means of the future variable f. Namely, at the **await**, if the task spawned at L37 has not finished, the processor is released and any available task on the This object could take it. The result of the execution of retrieveCoins is obtained by means of the blocking **get** instruction which *blocks* the execution of This until the future variable f is ready. In general, the use of **get** can introduce deadlocks. In this case, the **await** at L38 ensures that retrieveCoins has finished and thus the execution will not block.

Points-to Analysis. Inferring the set of memory locations to which a reference variable *may* point-to is a classical analysis in object-oriented languages. In SACO we follow Milonava et al. [11] and abstract objects by the *sequence of allocation sites* of all objects that lead to its creation. E.g., if we create an object o_1 at program point pp_1, and afterwards call a method of o_1 that creates an object o_2 at program point pp_2, then the abstract representation of o_2 is $pp_1.pp_2$. In order to ensure termination of the inference process, the analysis is parametrized by k, the maximal length of these sequences. In the example, for any $k \geq 2$, assuming that the allocation site of the This object is ϵ, the points-to analysis abstracts v and out to $\epsilon.35$ and $\epsilon.34$, respectively. For $k = 1$, they would be abstracted to 35 and 34. As variables can be reused, the information that the analysis gives is specified at the program point level. Basically, the analysis results are defined by a function $\mathcal{P}(o_p, pp, v)$ which for a given (abstract) object o_p, a program point pp and a variable v, it returns the set of abstract objects to which v may point to. For instance, $\mathcal{P}(\epsilon, 36, v) = 35$ should be read as: when executing This and instruction L36 is reached, variable v points to an object whose allocation site is 35. Besides, we can trivially use the analysis results to find out to which task a future variable f is pointing to. I.e., $\mathcal{P}(o_p, pp, f) = o.m$ where o is an abstract object and m a method name, e.g., $\mathcal{P}(\epsilon, 37, f) = 35.\mathsf{retrieveCoins}$. Points-to analysis allows making any analysis object-sensitive [11]. In addition, in SACO we use it: (1) in the resource analysis in order to know to which object the cost must be attributed, and (2) in the deadlock analysis, where the abstraction of future variables above is used to spot dependencies among tasks.

May-Happen-in-Parallel. An MHP analysis [10, 3] provides a safe approximation of the set of pairs of statements that can execute in parallel across several objects, or in an interleaved way within an object. MHP allows ensuring absence of data races, i.e., that several objects access the same data in parallel and at least one of them modifies such data. Also, it is crucial for improving the accuracy of deadlock, termination and resource analysis. The MHP analysis implemented in SACO [3] can be understood as a function $\mathcal{MHP}(o_p, pp)$ which returns the set of program points that may happen in parallel with pp when executing in the abstract object o_p. A remarkable feature of our analysis is that it performs a local analysis of methods followed by a composition of the local results, and it has a polynomial complexity. In our example, SACO infers that the execution of showIncome (L2) cannot happen in parallel with any instruction in retrieveCoins

(L18–L27), since retrieveCoins must be finished in the **await** at L38. Similarly, it also reveals that showCoin (L3) cannot happen in parallel with showIncome. On the other hand, SACO detects that the **await** (L24) and the assignment (L16) may happen in parallel. This could be a problem for the termination of retrieveCoins, as the shared variable coins that controls the loop may be modified in parallel, but our termination analysis can overcome this difficulty. Since the result of the MHP analysis refines the control-flow, we could also consider applying the MHP and points-to analyses continuously to refine the results of each other. In SACO we apply them only once.

3 Advanced Analyses

Termination Analysis. The main challenge is in handling *shared-memory* concurrent programs. When execution interleaves from one task to another, the shared-memory may be modified by the interleaved task. The modifications can affect the behavior of the program and change its termination behavior and its resource consumption. Inspired by the rely-guarantee principle used for compositional verification and analysis [12, 5] of thread-based concurrent programs, SACO incorporates a novel termination analysis for concurrent objects [4] which assumes a *property* on the global state in order to prove termination of a loop and, then, proves that this property holds. The property to prove is the *finiteness* of the shared-data involved in the termination proof, i.e., proving that such shared-memory is updated a finite number of times. Our analysis is based on a circular style of reasoning since the finiteness assumptions are proved by proving termination of the loops in which that shared-memory is modified. Crucial for accuracy is the use of the information inferred by the MHP analysis which allows us to restrict the set of program points on which the property has to be proved to those that may actually interleave its execution with the considered loop.

Consider the function retrieveCoins from Sec. 2. At the **await** (L24) the value of the shared variable coins may change, since other tasks may take the object's lock and modify coins. In order to prove termination, the analysis first assumes that coins is updated a finite number of times. Under this assumption the loop is terminating because eventually the value of coins will stop being updated by other tasks, and then it will decrease at each iteration of the loop. The second step is to prove that the assumption holds, i.e., that the instructions updating coins are executed a finite number of times. The only update instruction that may happen in parallel with the **await** is in insertCoin (L16), which is called from insertCoins and this from main. Since these three functions are terminating (their termination can be proved without any assumption), the assumption holds and therefore retrieveCoins terminates. Similarly, the analysis can prove the termination of the other functions, thus proving the whole program terminating.

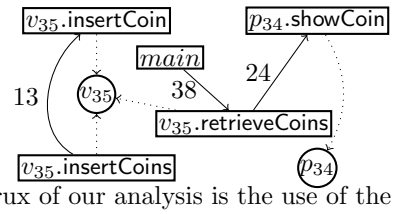
Resource Analysis. SACO can measure different types of costs (e.g., number of execution steps, memory created, etc.) [2]. In the output, it returns upper bounds on the worst-case cost of executing the concurrent program. The results

of our termination analysis provide useful information for cost: if the program is terminating then the size of all data is bounded (we use x^+ to refer to the maximal value for x). Thus, we can give cost bounds in terms of the maximum and/or minimum values that the involved data can reach. Still, we need novel techniques to infer upper bounds on the number of iterations of loops whose execution might interleave with instructions that update the shared memory. SACO incorporates a novel approach which is based on the combination of *local* ranking functions (i.e., ranking functions obtained by ignoring the concurrent interleaving behaviors) with upper bounds on the *number of visits* to the instructions which update the shared memory. As in termination, the function $\mathcal{MHP}$ is used to restrict the set of points whose visits have to be counted to those that indeed may interleave.

Consider again the loop inside retrieveCoins. Ignoring concurrent interleavings, a local ranking function RF = coins is easily computed. In order to obtain an upper bound on the number of iterations considering interleavings, we need to calculate the number of visits to L16, the only instruction that updates coins and MHP with the **await** in L24. We need to add the number of visits to L16 for every path of calls reaching it, in this case main–insertCoins–insertCoin only. By applying the analysis recursively we obtain that L16 is visited n times. Combining the local ranking function and the number of visits to L16 we obtain that an upper bound on the number of iterations of the loop in retrieveCoins is $\mathsf{coin}^+ * \mathsf{n}$.

Finally, we use the results of points-to analysis to infer the cost at the level of the distributed components (i.e., the objects). Namely, we give an upper bound of the form $c(\epsilon)^*(\dots)+c(35)^*(\mathsf{coin}^+ * \mathsf{n}.\dots)+ c(34)^*(\dots)$ which distinguishes the cost attributed to each abstract object o by means of its associated marker $c(o)$.

Deadlock Analysis. The combination of non-blocking (**await**) and blocking (**get**) mechanisms to access futures may give rise to complex deadlock situations. SACO provides a rigorous formal analysis which ensures deadlock freedom, as described in [6]. Similarly to other deadlock analyses, our analysis is based on constructing a *dependency graph* which, if acyclic, guarantees that the program is deadlock free. In order to construct the dependency graph, we use points-to analysis to identify the set of objects and tasks created along any execution. Given this information, the construction of the graph is done by a traversal of the program in which we analyze **await** and **get** instructions in order to detect possible deadlock situations. However, without further *temporal* information, our dependency graphs would be extremely imprecise. The crux of our analysis is the use of the MHP analysis which allows us to label the dependency graph with the program points of the synchronization instructions that introduce the dependencies and, thus, that may potentially induce deadlocks. In a post-process, we discard *unfeasible* cycles in which the synchronization instructions involved in the circular dependency cannot happen in parallel. The dependency graph for our example is shown above. Circular nodes represent objects and squares tasks. Solid edges are



tagged with the program point that generated them (**await** or **get** instructions). Dotted edges go from each task to their objects indicating ownership. In our example, there are no cycles in the graph. Thus, the program is deadlock free.

4 Related Tools and Conclusions

We have presented a powerful static analyzer for an actor-based concurrency model, which is lately regaining much attention due to its adoption in Erlang, Scala and concurrent objects (e.g., there are libraries in Java implementing concurrent objects). As regards to related tools, there is another tool [7] which performs deadlock analysis of concurrent objects but, unlike SACO, it does not rely on MHP and points-to analyses. We refer to [3, 6] for detailed descriptions on the false positives that our tool can give. Regarding termination, we only know of the TERMINATOR tool [8] for thread-based concurrency. As far as we know, there are no other cost analyzers for imperative concurrent programs.

Acknowledgments. 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

1. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *APLAS'11*, LNCS 7078, pp. 238-254. Springer, 2011.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *FORTE'12*, LNCS 7273, pp. 35-51. Springer, 2012.
4. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA'13*, LNCS 8172, pp. 349-364. Springer, 2013.
5. B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *PLDI'07, pp. 320–330.* ACM, 2007.
6. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS 7892, pp. 273-288. Springer, 2013.
7. E. Giachino and C. Laneve. Analysis of Deadlocks in Object Groups. In *FMOOD-S/FORTE*, LNCS 6722, pp. 168-182. Springer, 2011.
8. http://research.microsoft.com/en us/um/cambridge/projects/terminator/.
9. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO'10 (Revised Papers)*, LNCS 6957, pp. 142-164. Springer, 2012.
10. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *PPoPP'10, pp. 25-36.* ACM, 2010.
11. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.
12. C. Popeea and A. Rybalchenko. Compositional Termination Proofs for Multi-Threaded Programs. In *TACAS'12*, LNCS 7214, pp. 237-251. Springer, 2012.