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

Deliverable D2.2.2
Formalisation of Service Contracts and SLAs (Final Report)

Date of document: T30

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

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

Draft Revision March 28, 2016

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:
Formalisation of Service Contracts and SLAs (Final Report)

This document summarises deliverable D2.2.2 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`.

This deliverable reports the final outcome of Task T2.2, where the gap between (parts of) SLAs and services is bridged by

(i) developing a formal language for modelling SLA documents,

(ii) providing behavioural interfaces with quality of services descriptions that address virtualised resources and deployment models and

(iii) defining techniques to assess the compatibility between SLAs and service contracts.

List of Authors

Frank de Boer (CWI)
Elena Giachino (BOL)

Contents

Chapter 1

Introduction

1.1 Objectives

In Cloud Services and in Web Services, in general, resource provisioning is defined by means of legal contracts agreed upon by service providers and customers, called *service level agreements* – SLA. Legal contracts usually include measurement methods and scales that are used to set the boundaries and margins of errors that apply to the behaviour of the service, as well as the legal requirements under different jurisdictions. The SLA documents have no standardised format nor terminology, and do not abide by any precise definition, notwithstanding some recent attempts towards standardisation – see [2] and the references therein.

Because of this informal nature, there is a significant gap between SLAs and the corresponding services whose quality levels they constrain. As a consequence, SLAs are currently not integrated in the software artefacts, and assessing whether a service complies with an SLA or not is always a point of concern. As a consequence, providers very often over-provide resources to services, in order to avoid legal disputes, with the result of wasting resources and making services more expensive.

This deliverable, as anticipated in the Envisage DOW, reports the final outcome of Task T2.2, where the gap between (parts of) SLAs and services is bridged by

(i) developing a formal language for modelling SLA documents,

(ii) providing behavioural interfaces with quality of services (QoS) descriptions that address virtualised resources and deployment models, and

(iii) defining techniques to assess the compatibility between SLAs and service contracts.

To reach goal (i) we make use of simple formal descriptions of SLAs in terms of *metric functions* suitable to cover the specific Envisage case studies. We also address the (re)design, the (re)negotiation and/or termination, and the monitoring of SLAs within the dynamic context of changing business objectives and resource availability. Regarding goals (ii) and (iii), we define a mathematical framework that is able either to derive the SLA quality levels from the service programs and to verify possible violations, or to monitor service behaviours and document SLA quality level mismatches. In particular, we provide behavioural interfaces that take into account notions such as performance, delivery time, usage of computing resources, etc. Goal (iii) will precisely define the part of an SLA that may be statically verified to be compatible with a service contract, and the part that must be enforced by ad hoc monitoring add-ons defined in Task T2.3. Additionally, given the executable models of ENVISAGE services, all the above goals will be amenable to automatic verification and validation using the analysis techniques developed in WP3.

Among the properties whose qualities are constrained by SLA documents [12], we focus on *performance* by analysing the objectives that set the boundaries and margins of errors of service's behaviours. In Section 2, these objectives are formalised in terms of metric functions. Having at hand these functions, we address the problem to verify whether a given service complies with them or not. Two techniques are developed for verifying performance properties of services: the *static* techniques (discussed in Section 3) and the *runtime* ones (which are responsibility of D2.3.2).

In static techniques, the compliance of a service with respect to a metric function is shown by means of analysis tools that either directly verify the code (static analysis), or an underlying mathematical model (model checking, simulation, etc.). Whenever the service does not comply with the metric function, the designer triggers a sequence of code refinements that lead to compliance. As an example, consider *resource capacity* that measures how much a critical resource is used by a service. Section 3 reports a static analysis technique that uses so-called behavioural types. These behavioural types are abstract descriptions of programs that support compositional reasoning and that retain the necessary information to derive resource usage. By means of behavioural types, we use either a cost equation evaluator – the solver systems [8, 1] – or a theorem prover – the KeY system [3] – to prove compliance with the SLA. For instance, we demonstrate that the response time of a given method does not exceed a certain user-defined threshold.

In runtime techniques, the enforcement of properties is accomplished by using code that is external to the service and that continuously monitors it. In fact, there are (performance) metric functions that cannot be (even in principle) fully verified statically, due to factors under external control, such as the requests per minute by end users and failing machines in the underlying infrastructure. As an example, consider the percentage of successful requests, namely the number of requests processed by the service without a failure due to its infrastructure over the total number of received requests. In D2.3.2, we will report a technique based on an external monitoring system that filters service's replies, counts them, and records the erroneous ones. The correctness of the composite system consisting of the service and monitoring code is established by means of either static analysis techniques or model checking.

Figure 1.1 describes the flow of analysis techniques used in our approach. A *feedback loop* ensures corrections and improvements to the system. In particular, if the static analysis reports that a service does

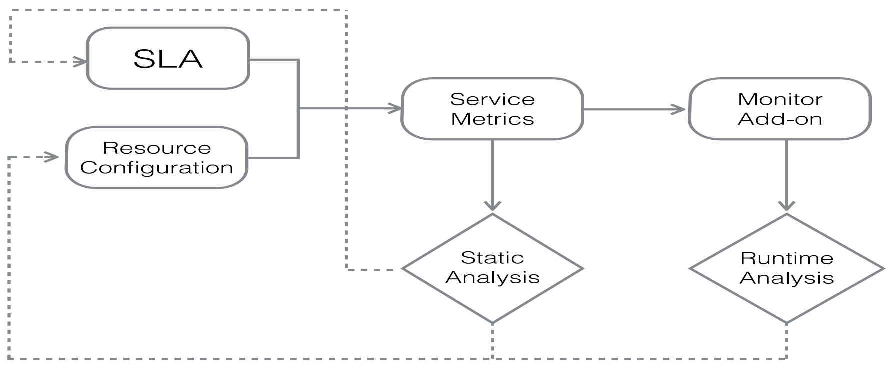

Figure 1.1: Analysis Flow: *Resource Configuration* refers to the configuration of resource types that are used for the service; *Service Metrics* denotes the set of metrics that define the quality of the service. The *dashed* lines present a feedback loop to a previous phase of analysis.

not match an SLA constraint, then, during the negotiation phase that constraint can be either relaxed or the resource configuration can be extended accordingly (with a possible charge for the client). Similarly, if a monitoring system verifies a runtime violation of an SLA constraint then, in order to avoid expensive penalties, the service providers trigger the resource configuration system to increase the resources of the services.

1.2 List of Papers Comprising Deliverable D2.2.2

This section lists all the papers that this deliverable comprises, indicates where they were published, and explains how each paper is related to the main text of this deliverable. The full papers are made available in the appendix of this deliverable and on the Envisage web site at the url `http://www.envisage-project.eu/` (select "Dissemination").

Paper 1: Statically and Dynamically verifiable SLA metrics There is a gap between runtime service behaviours and the contracted quality expectations with the customers that is due to the informal nature of service level agreements. This paper explains how to bridge the gap by formalising service level agreements with metric functions. We therefore discuss an end-to-end analysis flow that can either statically verify whether a service code complies with a metric function or use runtime monitoring systems to report possible misbehaviours. In both cases, our approach provides a feedback loop to fix and improve the metrics and eventually the resource configurations of the service itself.

The paper was written by Elena Giachino, Stijn de Gouw, Cosimo Laneve and Behrooz Nobakht. The paper is accepted and will appear in the book *"Theory and Practice of Formal Methods"* (volume 9660 of the Lecture Notes on Computer Science).

Paper 2: Static analysis of cloud elasticity This paper proposes a *static analysis technique* that computes upper bounds of virtual machine usages in a ABS-like language with explicit acquire and release operations of virtual machines. In particular, the language admits delegation of virtual machine releasing operations (by passing them as arguments of invocations). The technique is modular and consists of (*i*) a type system associating programs with behavioural types that records relevant information for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioural types and returns cost equations, and (*iii*) an automatic solver for the the cost equations.

The paper was written by Abel Garcia, Cosimo Laneve and Michael Lienhardt. The paper has been presented at the conference PPDP 2015 and appeared in the proceedings of the conference. A longer and revised version with proofs has been submitted to a journal.

Paper 3: Time complexity of concurrent programs This paper presents our approach to the problem of automatically computing the time complexity of concurrent object-oriented programs. To determine this complexity we use intermediate abstract descriptions that record relevant information for the time analysis (cost of statements, creations of objects, and concurrent operations), by means of behavioural types. Then, we define a translation function that takes behavioural types and makes the parallelism explicit into so-called cost equations, which are fed to an automatic off-the-shelf solver for obtaining the time complexity.

The paper was written by Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, and Ka I Pun. The paper has been presented at the conference FACS 2015 and appeared in the post-proceedings of the conference.

Chapter 2

Formalisation of SLA Metrics

In the "Cloud Service Level Agreement Standardisation Guidelines" document [12], the qualities of services are assessed with SLAs according to the properties they have, which range from performance to security and to data management. In this deliverable, we focus on *performance*. We discuss how performance can be formalised and evaluated on the source code level.

2.1 Performance Metrics, Informally

The article [12] distinguishes three kinds of performance properties: *availability*, *response time*, and *capacity*. Availability is the property of a service to be accessible and usable on demand. By detailing the notion of "usability", one gets different instances of availability and corresponding service metrics. For instance (*i*) *level of uptime*, is the time in a defined period the service is up, over the total possible available time; (*ii*) *percentage of successful requests*, is the number of requests processed without an error over the total number of submitted requests; (*iii*) *percentage of timely service provisioning requests*, is the number of service provisioning requests completed within a defined time period over the total number of service provisioning requests. Response time is the time period between a client request event and a service response event. The service metrics that are used to constrain response time may return either an *average time* or a *maximum time*, given a particular kind of request. Capacity is the maximum amount of some resource used by a service. It also includes the service throughput metric, namely the minimum number of requests that can be processed by a service in a stated time period.

The example below is taken from the FRH case study and discusses its corresponding SLA constraints about performance. The next section formalises the involved metrics and we show how to verify/enforce them in the rest of the deliverable.

Example 1 *SDL-Fredhopper offers search and targeting facilities on large product databases over cloud computing architectures to e-commerce companies. The offered services are exposed at endpoints and are typically implemented to accept connections over HTTP. For example, a* query API *allows users to query over a product catalog. Assume that the query API is implemented by means of a number of resources (virtual machines) that are managed in a mutual exclusive way by a load balancer (each resource is launched to serve* exactly one *instance of the query API). When an e-commerce company signs the SLA contract with the Cloud Service company, the performance properties of the query API are constrained by the following metrics:*

- 95% of requests is completed within 1 minute, 2 additional percentage points within 3 minutes and 1 additional percentage points within 5 minutes. *This is the "percentage of timely service provisioning requests" metric and it is used by the operations team of the Cloud Service company to set up an environment for the customer that includes the necessary resources to match the constraints. It is additionally used by the support team of the Cloud Service company to manage communications with the customer during the lifetime of the service for the customer.*

- the service completes 8 queries per minute from 9:00 to 18:00 and 4 queries per minute otherwise. *This is a service throughput metric and forms the basis of many decisions (technical or legal) thereafter, such as the definition of the necessary resources for the e-commerce company.*
- the service replies to a query request (with the result or with a failure) within 7 minutes. *This is a response time metric and may be determined by the database size as well as by the size of the data managed by the query service.*

2.2 Performance Metrics, Formally

To determine the precise level of a metric, and verify whether the service matches the agreed levels, an indisputable formalisation is needed, rather than the informal descriptions in the previous section. There have been several attempts to formalise SLAs, using techniques ranging from semantic annotations [14], to rewriting logics [16] and to constraint programming [4]. In this deliverable, following [15], we present a very simple formalisation based on *service metric functions.*

Service metric functions aggregate a set of basic measurements into a single number that indicates the quality of a certain service characteristic. For instance $\mu(\tau)$ and $\nu(\tau, \delta)$ are two functions that respectively take one and two inputs, where

- τ is an interval of the form $[\mathtt{d}.t,\ \mathtt{d}'.t']$, where $\mathtt{d}$, $\mathtt{d}'$ are days ($\mathtt{d}, \mathtt{d}' \in \{1, \dots, 366\}$) and t, t' are seconds in the day ($t, t' \in \{0, \dots, 86399\}$);
- δ can be an upper bound to the size in bytes of client's requests, a time bound for getting a reply, or an upper bound to the number of resources used by the service.

We formalise now the informally defined performance metrics from Example 1. In particular,

- the percentage of timely service provisioning requests of a service s can be formalised by the following function $\mathtt{PTS}_s$:

$$\mathtt{PTS}_s([1.0,\ 366.86399], x) \ = \ \begin{cases} 0,95 & \text{if } x = 60s \\ 0,97 & \text{if } x = 180s \\ 0,98 & \text{if } x = 300s \end{cases}$$

- the service throughput of a service s can be formalised by the function $\mathtt{ST}_s$ as follows:

$$\mathtt{ST}_s([1.t,\ 366.t'], 60) \ = \ \begin{cases} 4 & \text{if } t = 0 \text{ and } t' = 32399 \\ 8 & \text{if } t = 32400 \text{ and } t' = 64800 \\ 4 & \text{if } t = 64801 \text{ and } t' = 86399 \end{cases}$$

- the response time of a service s can be defined by the following function $\mathtt{RT}_s$:

$$\mathtt{RT}_s([1.0,\ 366.86399]) \ = \ 420s$$

2.3 Composite Metrics

SLA documents may contain (performance) metrics that are not directly defined in terms of those described so far but are a composition of them. We discuss an example taken from the ATB case study.

Example 2 *A mobile search app provides mobile offline search by means of on-device search indices that are built and distributed by a cloud service. A primary motivations for mobile offline search, besides increasing search availability and strengthen user privacy, is to reduce search latency by using consistently fast on-device storage rather than accessing mobile and Wi-Fi network with highly variable latency. As a consequence, the most relevant aspect for evaluating the quality of the provided service is the* freshness of index data on the mobile device. *This property specifies time-related guarantees about the interval between the publication of a document in the cloud and its indexing and availability on the mobile device.*

The metric freshness of index data on the mobile device, noted $\mathtt{FID}$, actually is the sum of the response time $\mathtt{RT}_s$ and the delivery time $\mathtt{DT}_s$, namely the time to transfer the data to the devices. This last metric $\mathtt{DT}_s$ depends on the data size of the response and the available bandwidth. While the data size δ is a parameter, the *bandwidth* metric $\mathtt{B}(\tau)$ is another basic *capacity* metric. $\mathtt{B}(\tau)$ is expressed in Mb/s and defines the minimum amount of bandwidth required by the service in a particular time frame. It turns out that $\mathtt{DT}_s(\tau, \delta) = \delta / B(\tau)$ and, therefore, we may define

$$\mathtt{FID}(\tau, \delta) = \mathtt{DT}_s(\tau, \delta) + \mathtt{RT}_s(\tau, \delta)\ .$$

2.4 Specifying Service Metric Functions with Attribute Grammars

In Section 2.2 we proposed to bridge the gap between SLAs and services through service metric functions. Intuitively, a service metric function measures the level of a QoS property of the associated service. To answer the question how service metric functions can be made amenable to analyses we refer to the work to be presented in D2.3.2 where we propose to formalize a service metric function as an attribute of an attribute grammar [13].

In general, attributes in a grammar are functions that map words of a language to a value. In our setting, the events published by the monitoring framework form the terminals of the grammar and the traces of such events (such as service invocations) are the words. The non-terminals of the grammar specify a protocol over these events, determining the order in which the events should occur. This can be used to detect invalid uses of the services. The value of an attribute is defined in the grammar productions, using the functional data types of ABS. This ensures that the value of the attribute is computable, and that their computation generates no side-effects that change the state of the underlying system/services.

We successfully applied attribute grammars in a runtime checker, focusing on dynamic deadlock detection and general functional properties. For more detailed information, and several example attribute grammars that illustrate these idea's, we refer to [6] and [5].

We will report in Deliverable D2.3.2 on the concrete specification of service metric functions in ABS (rather than only mathematical) using attributes in a grammar. The service metric functions can then be used in the formalization of SLAs. In the same deliverable we will also report on the (automatic) generation of monitoring add-ons from the grammars. The monitors will be ordinary ABS code. This in principle also allows leveraging static analyses: simply analyze the original system together with the generated ABS monitors.

Chapter 3

Behavioural Interfaces for Performance

Several static analysis techniques are possible in order to verify service properties and, in particular, service metrics like response time. In this section we discuss two approaches we use in the Envisage Project and we apply them to the response time metric of Example 1. We refer to the papers [9, 11] (corresponding to papers in Appendices B and C) for further details on the technique described in Section 3.1. We refer to [3, 7] for details on the technique discussed in Section 3.3.

3.1 Behavioural Types

Behavioural types are abstract descriptions of programs that highlight the relevant information to derive a particular property. This derivation usually consists of three steps:

1. an inference system parses the service program and returns a behavioural type;
2. the behavioural types are translated into low-level descriptions that are adequate for a solver;
3. the low-level descriptions are fed to a solver which produces the output.

It turns out that behavioural types support compositional reasoning and are therefore adequate for SLA compliance, while low-level descriptions are not compositional (and too intensional).

In case of response time analysis, the behavioural types carry information about the cost of operations that are extracted directly from the source program. This means that the source program retains either resource-consumption annotations or resource-aware commands. The following code snippets use explicit primitives for expressing the consumption of resources. In particular, the statement `job(e)` specifies a requirement of `e` CPU resources and is instrumental for modelling the time: depending on the available resources its execution might take an observable amount of time proportional to its cost. For instance, the execution of `job(6)` when only 3 CPU resources are available will be executed within 6/3=2 units of time.

We illustrate our technique with two examples derived from Example 1. We assume a simple setting where every instance runs in the same machine with a fixed *capacity* of `c` CPU resources.

Consider the service that performs a query on a database, in Figure 3.1. The method `searchDB` sends a given query to the database and, when the result of the query is returned, it enhances the result with some information before returning it to the client. The `job(h)` statement specifies that the local operations of `searchDB` require `h` CPU resources. The `query` method, which is implemented in a different class `DataBase`, receives a query, evaluates it, searches the corresponding item in the database, and returns the result. The overall cost for these operations is `k` CPU resources, as specified by `job(k)`. In this example we assume the methods `elaborate` and `search` contain no `job` statements, thus they do not require any resources. Their resource requirements are part of the `k` resources declared for the `query` method.

An informal argument gives `(k+h)/c` as the total time required by `searchDB` to reply to a query, where `c` are the available CPU resources. This means that if we have a *ResponseTime* requirement of completing this

```
String searchDB(String s) {
  String u, v ;
  u = DB.query(s) ;
  job(h) ;
  v = this.add_info(u) ;
  return v; }
```

```
class DataBase {
  String query(String s) {
    String z = this.elaborate(s);
    String value = this.search(z);
    job(k) ;
    return value; }
  ... }
```

Figure 3.1: The service searchDB performing a query on a database.

method within a specific number of time units, then we are able to establish the minimum CPU resources of a configuration that complies with the SLA.

To formalise the above argument, we extract the program features that are relevant for the time analysis. The resulting descriptions are called *behavioural types* and primarily highlight cost annotations and method invocations. For example, the behavioural types of the above methods are

```
Service.searchDB(a[x], b[y]) { DataBase.query(b[y]) ⨾ h/x } : _

Service.addinfo(a[x]) {0} : _

DataBase.query(a[x]) {
  DataBase.query(a[x]) ⨾
  DataBase.elaborate(a[x]) ⨾
  DataBase.search(a[x])⨾
  k/x } : _

DataBase.elaborate(a[x]) {0} : _

DataBase.search(a[x]) {0} : _
```

where

- the parameter a[x] binds the this object identity to a and the available capacity to x; similarly, b[y] binds the object identity of the receiver of the query invocation to b and its allocated capacity to y;
- the cost h/x is due to the amount of CPU requested by job(h) and the available CPU resources x (similarly for k/x);
- the term _ is the time information corresponding to the returned value, which is in this case empty;
- the term 0 is the empty behaviour, meaning that no time units are consumed.

With the behavioural type specifications at hand, we use two techniques for deriving services' properties: one is completely automatic and uses solvers of cost equations, and another is semi-automatic (but more precise) and uses theorem provers. We discuss them in detail in the following two subsections.

3.2 The Cost Equation Solver

To evaluate behavioural types specifications, we translate them into so-called *cost equations*, which are suitable for solvers available in the literature [8, 1]. These cost equations are terms of the form:

$$m(\bar{x}) = \mathit{exp} \quad [\mathit{se}]$$

where m is a (cost) function symbol, *exp* is an expression that may contain (cost) function symbols applications. In some cases, more than one equation may be defined for the same function symbol: for instance the if-then-else statement has one equation for each branch. In this case, *se* is an expression representing the conditions under which the corresponding cost must be taken into account.

Basically, we translate behavioural types of methods into cost equations, where (i) method invocations are translated into function applications, and (ii) cost expressions occurring in the types are left unmodified. For example, the translations of the foregoing methods are:

```
searchDB($x$,$y$) = query($y$) + h/$x$ + addinfo($x$)
query($x$) = elaborate($x$) + search($x$) + k/$x$
addinfo($x$) = elaborate($x$) = search($x$ ) = 0
```

It is worth observing that, in this case, being $x = y =$ c, the solver returns (h+k)/c, as we anticipated previously.

Let us consider a variation of this example, where the service and the database run on different machines. In this case the configuration includes at least two different machines, let us call them m_s and m_d with respectively c_s and c_d allocated CPU resources. At the time of the creation of the service instance we can specify on which machine it will be deployed, by using a statement of the form:

```
Service service = new Service in m_s;
```

Analogously, for the database we have

```
Database database = new Database in m_d;
```

In this setting, all invocations on external machines are to be considered asynchronous, where the caller and the callee execute simultaneously, and the synchronisation occurs when the caller attempt to access the result of the invocation. The snippet of the method searchDB is therefore refined into the following code where the asynchronous invocation is noted with "!" instead of "." and Fut<String> is the type of a future String value.

```
String searchDB(String s) {
   String u, v ; Fut<String> w ;
   w = DB!query(s) ;
   job(h) ;
   u = w.get ;
   v = this.add_info(u) ;
   return(v);
}
```

The operation w.get explicitly synchronises the caller with the callee. In this case, the cost equations of the above methods are

```
searchDB($x$,$y$) = $max$(query($y$) , h/$x$) + addinfo($x$)
query($x$) = elaborate($x$) + search($x$) + k/$x$
addinfo($x$) = elaborate($x$) = search($x$ ) = 0
```

Being $x =\mathsf{c}_s$ and $y =\mathsf{c}_d$, the solver returns the total cost of max(h/c_s,k/c_d).

3.3 The KeY System

The are cases where the cost equations solver either fails to deliver a result or the result is so over-approximated that it becomes unusable. In particular, the cost equations $m(\bar{x}) = exp\ [se]$ that the solver takes as inputs are constrained by the fact that *se* is a boolean expression in a decidable fragment of Peano arithmetic – *presburger arithmetic* which admits only addition and multiplication by integer constants. Therefore, whenever behavioural types use expressions that are not written in presburger arithmetics, we extend them by manually adding preconditions and specifying costs and metrics in the postconditions.

We use a semi-interactive theorem prover called KeY [7], which uses symbolic execution to analyse programs. Properties are specified in KeY using *dynamic logic* [17] and are demonstrated using the *sequent calculus* [10]. It turns out that most proof steps (usually more than 99%) are automatically applied by the proof search strategies. Behavioural types plus KeY verification support a compositional analysis: each type can be analysed in isolation, on the basis of its own definition and only the contracts of the other methods – without knowledge of the underlying definition of the other behavioural types. This is not the case of cost equations that, once produced, are a monolithic, global specification.

KeY can be leveraged by following the steps below:

1. replace the cost expression c in method bodies by an assignment time = time+c;

2. add method contracts, specifying in the postcondition of each method the expected response time using the variable time and the capacities of machines;

3. prove the resulting instrumented program with KeY.

Applying these steps, to the example above, yields the following annotated behavioural types:

```
//@ ensures time == \old(time) + k/y + h/x;
Service.searchDB(a[x], b[y]) {
        DataBase.query(b[y]) ⨟ time = time + h/x
     } : _

//@ ensures time == \old(time);
Service.addinfo(a[x]) {0} : _

//@ ensures time == \old(time) + k/x;
DataBase.query(a[x]) {
        DataBase.query(a[x]) ⨟ DataBase.elaborate(a[x]) ⨟
        DataBase.search(a[x]) ⨟ time = time + k/x
     } : _

//@ ensures time == \old(time);
DataBase.elaborate(a[x]) {0} : _

//@ ensures time == \old(time);
DataBase.search(a[x]) {0} : _
```

For parallel programs with asynchronously executing threads, the above instrumentation might overestimate the actual time and cost consumed: it always sums the cost of tasks. In these cases, the behavioural type is x.m() ||| y.n(), rather than x.m(); y.n() (the operation " ||| " represents parallel composition). KeY derives the cost of x.m() ||| y.n() by taking the maximum of the costs of x.m() and of y.n().

A useful task that KeY supports is the formal proof that response times of a method are under a defined threshold. This is achieved by the same instrumentation discussed above. The only change needed is in the behavioural types of methods: one can adjust the postcondition with an assertion of the form $\text{time} < \text{d}$, where d is a symbolic threshold. This is shown in the contract below.

```
//@ ensures time < d;
Service.searchDB(a[x], b[y]) {
  ...
}
```

3.4 Further Discussion: Runtime Monitoring and Conflicting Metrics

A positive response of the static analysis is not enough to guarantee that all the service metrics will be satisfied by the service. Factors under external control, such as the underlying infrastructure, may affect the quality of service. Thus, in order to enforce service metrics that cannot be verified statically we use code external to the service that continuously monitors it. We are not going to discuss how the monitoring platform is defined, this will be reported in D2.3.2, but for the sake of discussing the whole analysis flow, we informally present a scenario in which monitoring plays a central role.

Let us consider again the metrics of Example 1. The static analysis gave an upper bound for searchDB response time of (k+h)/c time units. Letting the available amount of CPU resources be 2 and k=5 and h=10, then we have a response time of 7.5 seconds. This satisfies the $\mathtt{RT}_s$ metrics, since it is well below the maximum response time imposed by the SLA. Therefore the initial configuration of 2 CPU resources is found to be well suited for assuring the required QoS. Notice that, considering the time for executing a single request of searchDB, we can deduce that the $\mathtt{ST}_{\mathtt{searchDB}}$ value is indeed reasonable.

In addition, assume that a monitor, which observes the execution of the service, has not logged any entry where the response time is greater then 420 seconds – i.e. the response time is still matched.

However, the launch of a throughput monitor reports that only 4 requests are served per minute, which violates the SLA (requiring to serve 8 requests per minute during the day) because of latency problems for scheduling the requests or for connecting to the database. Henceforth, a reaction is triggered which requests to the monitoring platform and obtains a machine with 2 additional CPU resources. The service is moved on to the new machine and the throughput monitor does not find any violation anymore. However, during the night, half of the resources would have been sufficient for meeting the SLA requirement (which is only 4 requests per minute during the night). The customer is paying for unnecessary resources.

To overcome such issues, we consider an additional metric defining the *budget* for the service with respect to particular time windows:

$$\texttt{Budget}_{\texttt{searchDB}}([1.t,\ 366.t']) \;=\; \begin{cases} 40 & \text{if } t = 0 \text{ and } t' = 32399 \\ 80 & \text{if } t = 32400 \text{ and } t' = 64800 \\ 40 & \text{if } t = 64801 \text{ and } t' = 86399 \end{cases}$$

Namely, $\texttt{Budget}_{\texttt{searchDB}}$ specifies that, during the day, the customer is willing to pay up to 80, while only half for the night.

The static techniques may verify whether a service complies with $\texttt{Budget}_{\texttt{searchDB}}$ or not. In particular, an adequate budget is the cost of the minimum number of resources the program needs to execute, which is the cost of an upper bound of resources needed by the program. Taking CPUs as relevant resources and assuming that each CPU resource costs 10, then the analysis will approve $\texttt{Budget}_{\texttt{searchDB}}$, since the allocated money is enough to pay for 8 resources during the day and 4 during the night. However, a runtime CPU reallocation has been triggered by the throughput monitor. It turns out that the budget compliance is not met anymore because the expenses for the resource usage double the nightly budget. In this case, the budget_monitor reacts by requiring a deallocation of half of the CPU units during the night.

It is worth to notice that the allocations and deallocations required by a monitoring system may lead to a cyclic behaviour that does not reach any stable point. Therefore, in order to enforce stability, we also consider the notion of *service guarantee time*, namely the total amount of time from the start of the monitoring platform that a service is expected to meet its expectations of the SLA.

Bibliography

[1] 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.

[2] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In M. Ali Babar and Marlon Dumas, editors, *2nd Nordic Symposium on Cloud Computing & Internet Technologies (NordiCloud'13)*, pages 59–63. ACM Press, 2013.

[3] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[4] Maria Grazia Buscemi and Ugo Montanari. Qos negotiation in service composition. *J. Log. Algebr. Program.*, 80(1):13–24, 2011.

[5] Frank S. de Boer and Stijn de Gouw. Run-time checking multi-threaded java programs. In *SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings*, volume 9587 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2016.

[6] Frank S. de Boer and Stijn de Gouw. Run-time deadlock detection. In *ProCoS Workshop on Provably Correct Systems*, TO APPEAR.

[7] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer-Verlag, 2015.

[8] 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.

[9] Abel Garcia, Cosimo Laneve, and Michael Lienhardt. Static analysis of cloud elasticity. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 125–136. ACM, 2015.

[10] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[11] Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, and Ka I Pun. Time complexity of concurrent programs. To appear in Proceedings of FACS 2015, 2015.

[12] Cloud Select Industry Group. Cloud service level agreement standardisation guidelines, June 2014. Developed as part of the Commission's European Cloud Strategy. Available at `http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm?action=display&doc_id=6138`.

[13] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[14] David L. Martin, Mark H. Burstein, Drew V. McDermott, Sheila A. McIlraith, Massimo Paolucci, Katia P. Sycara, Deborah L. McGuinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10(3):243–277, 2007.

[15] Behrooz Nobakht, Stijn de Gouw, and Frank S. de Boer. Formal verification of service level agreements through distributed monitoring. In Schahram Dustdar, Frank Leymann Schahram, and Massimo Villari, editors, *Service Oriented and Cloud Computing – 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015*, Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2015.

[16] Joseph Okika. *Analysis and Verification of Service Contracts*. PhD thesis, Department of Computer Science, Aalborg University, 2010.

[17] Vaughan R. Pratt. Semantical considerations on floyd-hoare logic. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 109–121, 1976.

Glossary

QoS The degree to which a provided activity promotes customer satisfaction. For example, quality of service (QoS) technologies used in the electronic or telephone networking business typically assists in optimising network traffic management in order to improve the experience of network users.

SLA Contract between the customer (service consumer) and the service provider which defines (among other things) the minimal quality of the offered service, and the compensation if this minimal level is not reached.

Service metrics (functions) Introduced for formalising SLA, the service metric function aggregates basic measurements into a single number that indicates the quality of a certain service characteristic.

ABS Abstract Behavioural Specification language. An executable class-based, concurrent, object-oriented modelling language based on Creol, created for the HATS project.

Behavioural Interface The intended behaviour of programs such as functional behaviour and resource consumption can be expressed in the behavioural interface. Formal specifications of program behaviour is useful for precise documentation, for the generation of test cases and test oracles, for debugging, and for formal program verification.

Behavioural Type Abstract specification of a program's behaviour at runtime, used to perform specific analyses on the program, like resource consumption analysis.

Appendix A

Statically and Dynamically Verifiable SLA metrics

Statically and Dynamically verifiable SLA metrics[⋆]

Elena Giachino[1], Stijn de Gouw[2], Cosimo Laneve[1], and Behrooz Nobakht[2]

[1] Dep. of Computer Science and Engineering, University of Bologna – INRIA Focus
{elena.giachino,cosimo.laneve}@unibo.it
[2] SDL Fredhopper
{bnobakht,sgouw}@sdl.com

Abstract. There is a gap between run-time service behaviours and the contracted quality expectations with the customers that is due to the informal nature of service level agreements. We explain how to bridge the gap by formalizing service level agreements with metric functions. We therefore discuss an end-to-end analysis flow that can either *statically* verify if a service code complies with a metric function or use *run-time* monitoring systems to report possible misbehaviours. In both cases, our approach provides a feedback loop to fix and improve the metrics and eventually the resource configurations of the service itself.

1 Introduction

In Cloud Services and in Web Services, in general, resource provisioning is defined by means of legal contracts agreed upon by service providers and customers, called *service level agreements* – SLA. Legal contracts usually include measurement methods and scales that are used to set the boundaries and margins of errors that apply to the behaviour of the service, as well as the legal requirements under different jurisdictions. The SLA documents have no standardized format nor terminology, and do not abide by any precise definition, notwithstanding some recent attempts towards standardization – see [2] and the references therein.

Because of this informal nature, there is a significant gap between SLAs and the corresponding services whose quality levels they constrain. As a consequence, SLAs are currently not integrated in the software artefacts, and assessing whether a service complies with an SLA or not is always a point of concern. As a consequence, providers, in order to avoid legal disputes, very often over-provide resources to services with the result of wasting resources and making services more expensive.

This paper presents the approach taken in the EU Project Envisage [2] where the gap between (parts of) SLAs and services is bridged by (*i*) using simple formal descriptions of SLAs in terms of *metric functions* and by (*ii*) defining

[⋆] This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, http://www.envisage-project.eu.

a mathematical framework that is able either to derive the SLA quality levels from the service programs and to verify possible violations or to monitor service behaviours and document SLA quality levels mismatches.

Among the properties whose qualities are constrained by SLA documents [11], we focus in Section 2 on *performance* by analyzing the objectives that set the boundaries and margins of errors of service's behaviours. In Section 3, these objectives are formalized in terms of metric functions. Having at hand these functions, we address the problem to verify whether a given service complies with them or not. Two techniques are discussed in this paper for verifying performance properties of services: the *static-time* techniques and the *run-time* ones.

In static-time techniques, the compliance of a service with respect to a metric function is shown by means of analysis tools that either directly verify the code (static analysis), or an underlying mathematical model (model checking, simulation, etc.). Whenever the service does not comply with the metric function, the designer triggers a sequence of code refinements that lead to compliance. As an example, consider *resource capacity* that measures how much a critical resource is used by a service. Section 4 reports a static analysis technique that uses so-called behavioural types. These behavioural types are abstract descriptions of programs that support compositional reasoning and that retain the necessary information to derive resource usage. By means of behavioral types, we use either a cost equation evaluator – the solver systems [7,1] – or a theorem prover – the KeY system [3] – to prove compliance with the SLA. For instance, we demonstrate that the response time of a given method does not exceed a certain user-defined threshold.

In run-time techniques, the enforcement of properties is accomplished by using code that is external to the service and that continuously monitors it. In facts, there are (performance) metric functions that cannot be (even in principle) fully verified statically, due to factors under external control, such as the requests per minute by end users and failing machines in the underlying infrastructure. As an example, consider the percentage of successful requests, namely the number of requests processed by the service without a failure due to its infrastructure over the total number of received requests. In Section 5, we report a technique that uses an external monitoring system filtering service's replies, counts them, and records the erroneous ones. The correctness of the composite system consisting of the service and monitoring code is established by means of either static analysis techniques or model checking.

Figure 1 describes the flow of analysis techniques used in our approach. A *feedback loop* ensures corrections and improvements to the system. In particular, if the static analysis reports that a service does not match an SLA constraint, then, during the negotiation phase that constraint can be either relaxed or the resource configuration can be extended accordingly (with a possible charge for the client). Similarly, if a monitoring system verifies a run-time violation of an SLA constraint then, in order to avoid expensive penalties, the service providers trigger the resource configuration system to increase service's resources.

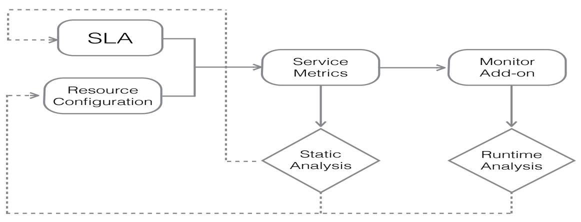

Fig. 1: Analysis Flow: *Resource Configuration* refers to the configuration of resource types that are used for the service; *Service Metrics* denotes the set of metrics that define the quality of the service. The *dashed* lines present a feedback loop to a previous phase of analysis.

In Section 6 we discuss the issue of SLA metrics that have conflicting requirements. In this case, it is necessary to determine an upper bound in time for reaching a stable resource configuration. We also discuss complex metrics that actually are compositions of basic metrics discussed in Section 3. We report our analysis of related works and conclude in Section 7.

2 SLAs and performance properties

In the "Cloud Service Level Agreement Standardisation Guidelines" document [11], the qualities of services are assessed with SLAs according to the properties they have, which range from performance to security and to data management. In this paper, we will focus on *performance*. We discuss how it can be formalized and evaluated on source code of services.

The article [11] distinguishes three kinds of performance properties: *availability*, *response time*, and *capacity*. Availability is the property of a service to be accessible and usable on demand. By detailing the notion of "usability", one gets different instances of availability and corresponding service metrics. For instance (*i*) *level of uptime*, is the time in a defined period the service is up, over the total possible available time; (*ii*) *percentage of successful requests*, is the number of requests processed without an error over the total number of submitted requests; (*iii*) *percentage of timely service provisioning requests*, is the number of service provisioning requests completed within a defined time period over the total number of service provisioning requests. Response time is the time period between a client request event and a service response event. The service metrics that are used to constrain response time may return either an *average time* or a *maximum time*, given a particular form of request. Capacity is the maximum amount of some resource used by a service. It also includes the service throughput metric, namely the minimum number of requests that can be processed by a service in a stated time period.

The example below discusses an industrial e-commerce use case and its corresponding SLA constraints about performance. The next section formalises the involved metrics and we show how to verify/enforce them in the rest of the paper.

Example 1. A Cloud Service company offers search and targeting facilities on large product databases over cloud computing architectures to e-commerce companies. The offered services are exposed at endpoints and are typically implemented to accept connections over HTTP. For example, a *query API* allows users to query over a product catalog. Assume that the query API is implemented by means of a number of resources (virtual machines) that are managed in a mutual exclusive way by a load balancer (each resource is launched to serve *exactly one* instance of the query API). When an e-commerce company signs the SLA contract with the Cloud Service company, the performance properties of the query API are constrained by the following metrics:

– *95% of requests is completed within 1 minute, 2% within 3 minutes and 1% within 5 minutes.* This is the "percentage of timely service provisioning requests" metric and it is used by the operations team of the Cloud Service company to set up an environment for the customer that includes the necessary resources to match the constraints. It is additionally used by the support team of the Cloud Service company to manage communications with the customer during the lifetime of the service for the customer.
– *the service completes 8 queries per minute from 9:00 to 18:00 and 4 queries per minute otherwise.* This is a service throughput metric and forms the basis of many decisions (technical or legal) thereafter, such as the definition of the necessary resources for the e-commerce company.
– *the service replies to a query request (with the result or with a failure) within 7 minutes.* This is a response time metric and may be determined by the size of database as well as by the size of the data managed by the query service (whenever the service accepts queries that are unbounded).

3 Metrics' formalization

To determine the precise level of a metric, and verify whether the service matches the agreed levels, an indisputable formalisation is needed, rather than the informal descriptions in the previous section. There have been several attempts to formalize SLAs, using techniques ranging from semantic annotations [17], to rewriting logics [19] and to constraint programming [5]. In this paper, following [18], we use a very simple formalization based on *service metric functions.*

Service metric functions aggregate a set of basic measurements into a single number that indicates the quality of a certain service characteristic. For instance $\mu(\tau)$ and $\nu(\tau, \delta)$ are two functions that respectively take one and two inputs, where

– τ is an interval of the form $[\mathtt{d}.t,\ \mathtt{d}'.t']$, where $\mathtt{d}, \mathtt{d}'$ are days ($\mathtt{d}, \mathtt{d}' \in \{1, \cdots, 366\}$) and t, t' are seconds in the day ($t, t' \in \{0, \cdots, 86399\}$);
– δ can be an upper bound to the size in bytes of client's requests, a time bound for getting a reply, or an upper bound to the number of resources used by the service.

To illustrate how performance metrics that are informally defined in SLA documents can be formalized, we further elaborate Example 1. In particular,

– the percentage of timely service provisioning requests of a service s can be formalized by the following function $\texttt{PTS}_s$:

$$\texttt{PTS}_s([1.0,\ 366.86399], x) \;=\; \begin{cases} 0,95 & \text{if } x = 60s \\ 0,97 & \text{if } x = 180s \\ 0,98 & \text{if } x = 300s \end{cases}$$

– the service throughput of a service s can be formalized by the function $\texttt{ST}_s$ as follows:

$$\texttt{ST}_s([1.t,\ 366.t'], 60) \;=\; \begin{cases} 4 & \text{if } t = 0 \text{ and } t' = 32399 \\ 8 & \text{if } t = 32400 \text{ and } t' = 64800 \\ 4 & \text{if } t = 64801 \text{ and } t' = 86399 \end{cases}$$

– the response time of a service s can be defined by the following function $\texttt{RT}_s$:

$$\texttt{RT}_s([1.0,\ 366.86399]) \;=\; 420s$$

4 Static-time analysis

Several static-time analysis techniques are possible to verify service properties and, in particular, service metrics like response time. In this section we discuss two approaches we use in the Envisage Project and we apply them to the response time metric of Example 1. We refer to [8,10] for further details on the technique described in Section 4.1. We refer to [3,6] for details on the technique discussed in Section 4.3.

4.1 Behavioural types

Behavioural types are abstract descriptions of programs that highlight the relevant informations to derive a particular property. This derivation usually consists of three steps:

1. an inference system parses the service program and returns a behavioural type;
2. the behavioural types are translated into low-level descriptions that are adequate for a solver;
3. the low-level descriptions are fed to a solver which produces the output.

It turns out that behavioural types support compositional reasoning and are therefore adequate for SLA compliance, while low-level descriptions are not compositional (and too intensional).

In the case of response time analysis, the behavioural types carry informations about costs of operations that are extracted directly from the source program. This means that the source program retains either resource-consumption

```
String searchDB(String s) {
  String u, v ;
  u = DB.query(s) ;
  job(h) ;
  v = this.add_info(u) ;
  return v; }
```

```
class DataBase {
  String query(String s) {
    String z = this.elaborate(s);
    String value = this.search(z);
    job(k) ;
    return value; }
  ... }
```

Fig. 2: The service `searchDB` performing a query on a database.

annotations or resource-aware commands. The following code snippets use explicit primitives for expressing the consumption of resources; in particular, the statement `job(e)` specifies a requirement of `e` CPU resources and is instrumental for modeling the time: depending on the available resources its execution might take an observable amount of time proportional to its cost. For instance, the execution of `job(6)` when only 3 CPU resources are available will be executed within 6/3=2 units of time.

We illustrate our technique with two examples derived from Example 1. We assume a simple setting where every instance runs in the same machine with a fixed *capacity* of `c` CPU resources.

Consider the service that performs a query on a database, in Figure 2. The method `searchDB` sends a given query to the database and, when the result of the query is returned, it enhances the result with some information before returning it to the client. The `job(h)` statement specifies that the local operations of `searchDB` require `h` CPU resources. The `query` method, which is implemented in a different class `DataBase`, receives a query, evaluates it, searches the corresponding item in the database, and returns the result. The overall cost for these operations is `k` CPU resources, as specified by `job(k)`. In this example we assume the methods `elaborate` and `search` contain no `job` statements, thus they do not require any resources. Their resource requirements are part of the `k` resources declared for the `query` method.

An informal argument gives `(k+h)/c` as the total time required by `searchDB` to reply to a query, where `c` are the available CPU resources. This means that if we have a *ResponseTime* requirement of completing this method within a specific number of time units, then we are able to establish the minimum CPU resources of a configuration that complies with the SLA.

To formalise the above argument, we extract the program features that are relevant for the time analysis. The resulting descriptions are called *behavioral types* and primarily highlight cost annotations and method invocations. For example, the behavioural types of the above methods are

```
Service.searchDB(a[x], b[y]) { DataBase.query(b[y]) ⨾ h/x
} : _

Service.addinfo(a[x]) {0} : _

DataBase.query(a[x]) { DataBase.query(a[x]) ⨾
  DataBase.elaborate(a[x]) ⨾ DataBase.search(a[x])
  ⨾ k/x } : _

DataBase.elaborate(a[x]) {0} : _
```

```
    DataBase.search(a[x]) {0} : _
```

where

- the parameter `a[x]` binds the `this` object identity to `a` and the available capacity to `x`; similarly, `b[y]` binds the object identity of the receiver of the `query` invocation to `b` and its allocated capacity to `y`;
- the cost `h/x` is due to the amount of CPU requested by `job(h)` and the available CPU resources `x` (similarly for `k/x`);
- the term `_` is the time information corresponding to the returned value, which is in this case empty;
- the term `0` is the empty behaviour, meaning that no time units are consumed.

With the behavioural type specifications at hand, we use two techniques for deriving services' properties: one is completely automatic and uses solvers of cost equations, and another is semi-automatic (but more precise) and uses theorem provers. We discuss them in detail in the following two subsections.

4.2 The cost equation solver

To evaluate behavioural types specifications, we translate them into so-called *cost equations*, which are suitable for solvers available in the literature [7,1]. These cost equations are terms

$$m(\overline{x}) = \mathit{exp} \quad [\mathit{se}]$$

where m is a (cost) function symbol, *exp* is an expression that may contain (cost) function symbols applications. In some cases, more than one equation may be defined for the same function symbol: for instance the if-then-else statement has one equation for each branch. In this case, *se* is an expression representing the conditions under which the corresponding cost must be taken into account.

Basically, we translate behavioural types of methods into cost equations, where (i) method invocations are translated into function applications, and (ii) cost expressions occurring in the types are left unmodified. For example, the translations of the foregoing methods are:

```
    searchDB(x,y) = query(y) + h/x + addinfo(x)
    query(x) = elaborate(x) + search(x) + k/x
    addinfo(x) = elaborate(x) = search(x ) = 0
```

It is worth to observe that, in this case, being $x = y =$`c`, the solver returns `(h+k)/c`, as we anticipated previously.

Let us consider a variation of this example, where the service and the database run on different machines. In this case the configuration will include at least two different machines, let us call them $\mathtt{m}_s$ and $\mathtt{m}_d$ with respectively $\mathtt{c}_s$ and $\mathtt{c}_d$ allocated CPU resources. At the time of the creation of the service instance we can specify on which machine it will be deployed, by using a statement of the form:

```
  Service service = new Service in m_s;
```

Analogously, for the database we have

```
Database database = new Database in m_d;
```

In this setting, all invocations on external machines are to be considered asynchronous, where the caller and the callee execute simultaneously, and the synchronization occurs when the caller attempt to access the result of the invocation. The snippet of the method `searchDB` is therefore refined into the following code where the asynchronous invocation is noted with "`!`" instead of "`.`" and `Fut<String>` is the type of a future `String` value.

```
String searchDB(String s) {
   String u, v ; Fut<String> w ;
   w = DB!query(s) ;
   job(h) ;
   u = w.get ;
   v = this.add_info(u) ;
   return(v);
}
```

The operation `w.get` explicitly synchronizes the caller with the callee. In this case, the cost equations of the above methods are

$$\begin{aligned}&\texttt{searchDB}(x,y) = max(\texttt{query}(y)\ ,\ \texttt{h}/x) + \texttt{addinfo}(x)\\&\texttt{query}(x) = \texttt{elaborate}(x) + \texttt{search}(x) + \texttt{k}/x\\&\texttt{addinfo}(x) = \texttt{elaborate}(x) = \texttt{search}(x\ ) = 0\end{aligned}$$

Being $x = \texttt{c}_s$ and $y = \texttt{c}_d$, the solver returns the total cost of $max(\texttt{h}/\texttt{c}_s, \texttt{k}/\texttt{c}_d)$.

4.3 The KeY system

The are cases where the cost equations solver either fails to deliver a result or the result is so over-approximated that it becomes unusable. In particular, the cost equations $m(\overline{x}) = exp\ [se]$ that the solver takes as inputs are constrained by the fact that se is a boolean expression in a decidable fragment of Peano arithmetic – *presburger arithmetic* which admits only addition and multiplication by integer constants. Therefore, whenever behavioural types use expressions that are not written in presburger arithmetics, we extend them by manually adding preconditions and in the postconditions specifying costs and metrics.

We use a semi-interactive theorem prover called KeY [6], which uses symbolic execution to analyze programs. Properties are specified in KeY using *dynamic logic* [20] and are demonstrated using the *sequent calculus* [9]. It turns out that most proof steps (usually more than 99%) are automatically applied by the proof search strategies. Behavioral types plus KeY verification support a compositional analysis: each type can be analyzed in isolation, on the basis of its own definition and only the contracts of the other methods – without knowledge of the underlying definition of the other behavioral types. This is not the case of cost equations that, once produced, are a monolithic, global specification.

KeY can be leveraged by following the steps below:

1. replace the cost expression `c` in method bodies by an assignment `time = time+c;`;
2. add method contracts, specifying in the postcondition of each method the expected response time using the variable `time` and the capacities of machines;

3. prove the resulting instrumented program with KeY.

Applying these steps yields the following annotated behavioral types:

```
//@ ensures time == \old(time) + k/y + h/x;
Service.searchDB(a[x], b[y]) {
       DataBase.query(b[y]) ⨟ time = time + h/x
     } : _

//@ ensures time == \old(time);
Service.addinfo(a[x]) {0} : _

//@ ensures time == \old(time) + k/x;
DataBase.query(a[x]) {
       DataBase.query(a[x]) ⨟ DataBase.elaborate(a[x]) ⨟
       DataBase.search(a[x]) ⨟ time = time + k/x
     } : _

//@ ensures time == \old(time);
DataBase.elaborate(a[x]) {0} : _

//@ ensures time == \old(time);
DataBase.search(a[x]) {0} : _
```

For parallel programs with asynchronously executing threads, the above instrumentation might overestimate the actual time and cost consumed: it always sums the cost of tasks. In these cases, the behavioural type is `x.m() ||| y.n()`, rather than `x.m(); y.n()` (the operation "`|||`" represents parallel composition). KeY derives the cost of `x.m() ||| y.n()` by taking the maximum of the costs of `x.m()` and of `y.n()`.

A useful task that KeY supports is the formal proof that response times of a method are under a defined threshold. This is achieved by the same instrumentation discussed above. The only change needed is in the behavioural types of methods: one can adjust the postcondition with an assertion of the form `time < d`, where `d` is a symbolic threshold. This is shown in the contract below.

```
//@ ensures time time < d;
Service.searchDB(a[x], b[y]) {
  ...
}
```

5 Run-time analysis

In order to enforce service metrics that cannot be verified statically (because of factors under external control, such as the underlying infrastructure) we use code external to the service that continuously monitors it. We discuss this technique using two service metrics of Example 1: the percentage of timely service provisioning requests and the service throughput.

A simple implementation of the function $\mathtt{PTS}_s$ defined in Section 3 uses a monitoring method that intercepts all the HTTP invocations to a service and their corresponding replies. This allows the monitor to record the time taken by every request to be completed. Consider the following pseudo-code for this method

```
void monitor_service_time() {
   (service,method,msg,client,m_id) = HttpRequest.intercept();
   time_start = time();
   reply = service.method(msg);
   time_end = time();
   HttpResponse.send(client,reply,m_id);
   log(m_id,time_start,time_end);
}
```

The method `percentage` takes as input a time window and returns `true` if the percentage of requests complies with the definition of $\texttt{PTS}_s$, is implemented by the monitor:

```
boolean percentage(Time t_begin, Time t_end){
    boolean v = true ;

    /* retrieve from the log the total number of messages
       served in the time window */
    nmb_msg = get_total_messages(t_begin, t_end) ;

    /* check whether the SLA percentages of served requests
       correspond to the observed ones */
     nmb_msg_completed = find(t_begin, t_end, 60) ;
     v = v && (nmb_msg_completed/nmb_msg <= 0.95) ; //95% in 1 min
     nmb_msg_completed = find(t_begin, t_end, 180) ;
     v = v && (nmb_msg_completed/nmb_msg <= 0.97) ; //97% in 3 mins
     nmb_msg_completed = find(t_begin, t_end, 300) ;
     v = v && (nmb_msg_completed/nmb_msg <= 0.98) ; //98% in 5 mins

    return v;
  }
```

Similarly, the monitor implementing the service metric $\texttt{ST}_s$ in Section 3 is the method:

```
boolean throughput(Log_file d, Time t_begin, Time t_end){
    int daily = 0;
    int nightly = 0;

    /* collects the number of the served requests during the two
       specified time-frames */
    for each (m_id, time_init, time_end) in d {
      if ((time_init >= 32400) && (time_init <= 64800)) // 9:00-18:00
            daily = daily + 1 ;
      else nightly = nightly + 1 ;
    }
    /* return true if 8 queries per minute are completed in 9:00-18:00
       and 4 queries per minute in the remaining time */

    return ( ((daily/60*9)>8) && ((nightly/60*15)>4) );
  }
```

The above straightforward development of monitoring systems allows service providers to report violations of the agreed SLA. However, the ultimate goal for a provider is to *maintain* the resource configuration in such a way that SLA violations remain under a given threshold while minimizing the cost of the system. The first objective can be achieved by adding resources to the service (for instance, adding more CPUs).

To this aim, the monitoring platform works in two cyclic phases: *observation* and *reaction*. The observation phase takes measurements on services – the foregoing methods `percentage` and `throughput`. Subsequently, if an SLA mismatch is observed, in the reaction phase, the number of allocated resources are increased. The monitoring platform developed in the Envisage Project also allows to *decrease* the number of resources if it is too costly/high [18]. The following `reaction` method verifies every 300s whether the percentage of timely service provisioning requests is reached and, in case of failures, adds one more CPU:

```
void reaction(Service s) {
   Time t ; Bool v ;
   t = time() ;
   idle(300) ;
   v = percentage(d,t, t+300) ;
   if (!v) MonitoringPlatform ! allocate(s) ;
}
```

Correctness of the monitoring framework (i.e. that the monitors converge within a user-given time towards the service level objectives specified in an SLA) was investigated in [18]. The idea is to translate the code for the program *including the monitoring code* into timed automata for use with UPPAAL [4]. The service level constraints from SLAs are translated into deadlines for the automata. The translation can be done automatically, along the lines of [12]. It is then possible to prove that, if all timed automata are schedulable (no missed deadline), then the SLA of the service is satisfied in the given timeframe.

6 Further aspects of metrics' definition and verification

In the previous sections we have discussed basic service metrics used in SLA documents. In this section we address two additional issues: (i) metrics may be conflicting: one metric requires an increase of resources allocated to a service, while another one requires a decrease of the same resources, and (ii) particular services may require complex service metrics.

Conflicting Metrics. Consider the following SLA constraints for the first example of Section 4.1:

$$\mathtt{ST}_{\mathtt{searchDB}}([1.t,\ 366.t'], 60) \ = \ \begin{cases} 4 & \text{if } t = 0 \text{ and } t' = 32399 \\ 8 & \text{if } t = 32400 \text{ and } t' = 64800 \\ 4 & \text{if } t = 64801 \text{ and } t' = 86399 \end{cases}$$

$$\mathtt{RT}_{\mathtt{searchDB}}([1.0,\ 366.86399]) \ = \ 420s$$

The analysis of Section 4.2 gave an upper bound for `searchDB` response time of `(k+h)/c` time units. Letting the available amount of CPU resources be 2 and `k=5` and `h=10`, then we have a response time of 7.5 seconds. This satisfies the $\mathtt{RT}_s$ metrics, since it is well below the maximum response time imposed by the SLA. Therefore the initial configuration of 2 CPU resources is found to be well suited for assuring the required QoS. Notice that, considering the time for executing a single request of `searchDB`, we can deduce that the $\mathtt{ST}_{\mathtt{searchDB}}$ value is indeed reasonable. In addition, assume that `monitor_service_time`, which observes the execution of the service, has not logged any entry where `time_end-time_begin` is greater then 420 seconds – i.e. the response time is still matched.

However, the launch of the `throughput` monitor reports that only 4 requests are served per minute, which violates the SLA (requiring to serve 8 requests per minute during the day) because of latency problems for scheduling the requests or for connecting to the database. Henceforth, the `reaction` method requests to the monitoring platform and obtains a machine with 2 additional CPU resources. The service is moved on the new machine and the `throughput` monitor doesn't find any violation anymore. However, during the night, half of the resources would have been sufficient for meeting the SLA requirement (which is only 4 requests per minute during the night). The customer is paying for unnecessary resources.

To overcome such issues, we consider an additional metric defining the *budget* for the service with respect to particular time windows:

$$\mathtt{Budget}_{\mathtt{searchDB}}([1.t,\ 366.t']) \ = \begin{cases} 40 & \text{if } t = 0 \text{ and } t' = 32399 \\ 80 & \text{if } t = 32400 \text{ and } t' = 64800 \\ 40 & \text{if } t = 64801 \text{ and } t' = 86399 \end{cases}$$

Namely, $\mathtt{Budget}_{\mathtt{searchDB}}$ specifies that, during the day, the customer is willing to pay up to 80, while only half for the night.

The techniques discussed in Section 4 may verify whether a service complies with $\mathtt{Budget}_{\mathtt{searchDB}}$ or not. In particular, an adequate budget is the cost of the minimum number of resources the program needs to execute, which is the cost of an upper bound of resources needed by the program. Taking CPUs as relevant resources and assuming that each CPU resource costs 10, then the analysis will approve $\mathtt{Budget}_{\mathtt{searchDB}}$, since the allocated money is enough to pay for 8 resources during the day and 4 during the night. However, a run-time CPU reallocation has been triggered by the `throughput` monitor. It turns out that the budget compliance is not met anymore because the expenses for the resource usage double the nightly budget. In this case, the `budget_monitor` reacts by requiring a deallocation of half of the CPU units during the night.

It is worth to notice that the allocations and deallocations required by a monitoring system may lead to a cyclic behaviour that does not reach any stable point. Therefore, in order to enforce stability, we also consider the notion of *service guarantee time*, namely the total amount of time from the start of the monitoring platform that a service is expected to meet its expectations of the SLA. In facts, we use the following refined version of `reaction` method of Section 5:

```
void reaction(Service s) {
    Time t ; Bool v ;
    t = time() ; idle(300) ;
    v = percentage(d, t, t+300) ;
    if (!v) {
           if (t > global_time_start + t_G) { // SLA is violated
                        notify(s, ‘‘SLA violation’’) ;
           } else MonitoringPlatform.allocate(s) ;
    }
}
```

Composite metrics. SLA documents may contain (performance) metrics that are not directly defined in terms of those in Section 3 but are a composition of them. We discuss an example.

Example 2. A mobile search app provides mobile offline search by means of on-device search indices that are built and distributed by a cloud service. A primary motivations for mobile offline search, besides increasing search availability and strengthen user privacy, is to reduce search latency by using consistently fast on-device storage rather than accessing mobile and Wi-Fi network with highly variable latency. As a consequence, the most relevant aspect for evaluating the quality of the provided service is the *freshness of index data on the mobile device.* This property specifies time-related guarantees about the interval between the publication of a document in the cloud and its indexing and availability on the mobile device.

The metric freshness of index data on the mobile device, noted $\mathtt{FID}$, actually is the sum of the response time $\mathtt{RT}_s$ and the delivery time $\mathtt{DT}_s$, namely the time to transfer the data to the devices. This last metric $\mathtt{DT}_s$ depends on the data size of the response and the available bandwidth. While the data size δ is a parameter, the *bandwidth* metric $\mathtt{B}(\tau)$ is another basic *capacity* metric (which has not been discussed in Section 3). $\mathtt{B}(\tau)$ is expressed in Mb/s and defines the minimum amount of bandwidth required by the service in a particular time frame. It turns out that $\mathtt{DT}_s(\tau, \delta) = \delta / B(\tau)$ and, therefore, we may define

$$\mathtt{FID}(\tau, \delta) = \mathtt{DT}_s(\tau, \delta) + \mathtt{RT}_s(\tau, \delta) \ .$$

7 Conclusions and related works

The methodology we have presented in this paper is being devised in the context of the EU Project Envisage [2]. The aim of the project is to develop a semantic foundation for virtualization and SLA that makes it possible to efficiently develop SLA-aware and scalable services, supported by highly automated analysis tools using formal methods. SLA-aware services are able to control their own resource management and renegotiate SLA across the heterogeneous virtualized computing landscape. The two examples we analyze in this contribution are taken from industrial case studies in the aforementioned project: the service described in Example 1 is an actual service provided by Fredhopper Cloud Services [3]. The mobile app presented in Example 2 is the Memkite app by Atbrox [4].

In the Envisage Project we also use other techniques for analyzing services, such as simulations and test generation covering critical scenarios. We intend to investigate if these additional techniques can be used for SLA compliance (and to what extent). For example, if they can provide augmented precision or more detailed descriptions of misbehaviours.

[3] `http://www.sdl.com/products/fredhopper/`
[4] `http://atbrox.com/`

Related Work. Several proposals define a language or a framework to formalize SLAs. However, there is no study how such SLAs can be used to verify or monitor the service and upgrade it as necessary. In this respect, up-to our knowledge, our technique that uses both static time analysis and run-time analysis is original.

As regards SLA formalizations, we recall few recent efforts. WSLA [14] introduces a framework that defines SLAs in a technical way and breaks down customer agreements in terms to be monitored. SLAng [15] introduces a language for defining metrics that deal with the *problems of networks* and studies a technique to ensure the corresponding service qualities. SLA* [13] introduces a generic language to specify SLAs with a fine-grained level of detail. In [16], a method is proposed to translate the SLA specification into an operational monitoring specification. This technique is being used by the EU Project SLA@SOI.

References

1. 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 *Proceedings of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer-Verlag, 2014.
2. Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In M. Ali Babar and Marlon Dumas, editors, *Proceedings of NordiCloud'13*, pages 59–63. ACM Press, 2013.
3. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
4. Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on Uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
5. Maria Grazia Buscemi and Ugo Montanari. Qos negotiation in service composition. *J. Log. Algebr. Program.*, 80(1):13–24, 2011.
6. Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In *Proceedings of CADE-25*, volume 9195 of *LNCS*, pages 517–526. Springer-Verlag, 2015.
7. 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 *LNCS*, pages 275–295. Springer, November 2014.
8. Abel Garcia, Cosimo Laneve, and Michael Lienhardt. Static analysis of cloud elasticity. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 125–136. ACM, 2015.
9. Gerhard Gentzen. Untersuchungen ber das logische schlieen. i. *Mathematische Zeitschrift*, 39(1):176–210, 1935.
10. Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, and Ka I Pun. Time complexity of concurrent programs. To appear in Proceedings of FACS 2015, 2015.
11. Cloud Select Industry Group. Cloud service level agreement standardisation guidelines, June 2014. Developed as part of the Commissions European Cloud Strategy. Available at `http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm?action=display&doc_id=6138`.

12. Mohammad Mahdi Jaghoori. Composing real-time concurrent objects refinement, compatibility and schedulability. In *Fundamentals of Software Engineering*, pages 96–111. Springer Berlin Heidelberg, 2012.
13. Keven T Kearney, Francesco Torelli, and Constantinos Kotsokalis. SLA$\star$: An abstract syntax for Service Level Agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224. IEEE, 2010.
14. Alexander Keller and Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.
15. D. Davide Lamanna, James Skene, and Wolfgang Emmerich. Slang: A language for defining service level agreements. In *Proccedings of (FTDCS'03)*, page 100. IEEE Computer Society, 2003.
16. Khaled Mahbub, George Spanoudakis, and Theocharis Tsigkritis. Translation of SLAs into monitoring specifications. In *Service Level Agreements for Cloud Computing*, pages 79–101. Springer, 2011.
17. David L. Martin, Mark H. Burstein, Drew V. McDermott, Sheila A. McIlraith, Massimo Paolucci, Katia P. Sycara, Deborah L. McGuinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10(3):243–277, 2007.
18. Behrooz Nobakht, Stijn de Gouw, and Frank S. de Boer. Formal verification of service level agreements through distributed monitoring. In Schahram Dustdar, Frank Leymann Schahram, and Massimo Villari, editors, *Proceedings of ESOCC 2015*, LNCS, pages 125–140. Springer-Verlag, 2015.
19. Joseph Okika. *Analysis and Verification of Service Contracts*. PhD thesis, Department of Computer Science, Aalborg University, 2010.
20. Vaughan R. Pratt. Semantical considerations on floyd-hoare logic. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 109–121, 1976.

Appendix B

Static Analysis of Cloud Elasticity

Static analysis of cloud elasticity *

Abel Garcia Cosimo Laneve Michael Lienhardt
Department of Computer Science and Engineering, University of Bologna – INRIA Focus
{abel.garcia2, cosimo.laneve, michael.lienhardt}@unibo.it

Abstract

We propose a *static analysis technique* that computes upper bounds of virtual machine usages in a *concurrent* language with explicit acquire and release operations of virtual machines. In our language it is possible to delegate other (ad-hoc or third party) concurrent code to release virtual machines (by passing them as arguments of invocations). Our technique is modular and consists of (*i*) a type system associating programs with behavioural types that records relevant information for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioural types and return cost equations, and (*iii*) an automatic off-the-shelf solver for the cost equations. A soundness proof of the type system establishes the correctness of our technique with respect to the cost equations. We have experimentally evaluated our technique using a cost analysis solver and we report some results. The experiments show that our analysis allows us to derive bounds for programs that are better than other techniques, such as those based on amortized analysis.

Categories and Subject Descriptors F.3.2 [*Logics and meanings of programs*]: Semantics of Programming Languages—Operational semantics,Program analysis ; F.1.1 [*Computation by abstract devices*]: Models of Computation—Relations between models

General Terms Static analysis, Resource consumption, Concurrent programming, Behavioural type system, Subject reduction.

Keywords Virtual machines creations and releases, transition relation, behavioural types, peak cost, net cost, cost equations.

1. Introduction

The analysis of resource usage in a program is of great interest because an accurate assessment could reduce energy consumption and allocation costs. These two criteria are even more important today, in modern architectures like mobile devices or cloud computing, where resources, such as virtual machines, have hourly or monthly rates. In fact, cloud computing introduces the concept of *elasticity*, namely the possibility for virtual machines to scale according to the software needs. In order to support elasticity, cloud providers, including Amazon, Google, and Microsoft Azure, (1) have pricing models that allow one to hire on demand virtual machine instances and paying them for the time they are in use, and (2) have APIs that include instructions for requesting and releasing virtual machine instances.

While it is relatively easy to estimate worst-case costs for simple code examples, extrapolating this information for fully real-life complex programs could be cumbersome and highly error-sensitive. The first attempts about the analysis of resource usage dates back to Wegbreit's pioneering work in 1975 [21], which develops a technique for deriving closed-form expressions out of programs. The evaluation of these expressions would return upper-bound costs that are parametrised by programs' inputs.

Wegbreit's contribution has two limitations: it addresses a simple functional languages and it does not formalize the connection between the language and the closed-form expressions. A number of techniques have been developed afterwards to cope with more expressive languages (see for instance [4, 9]) and to make the connection between programs and closed-form expressions precise (see for instance [10, 15]). A more detailed discussion of the related work in the literature is presented in Section 8.

To the best of our knowledge, current cost analysis techniques always address (concurrent) languages featuring only addition of resources. When removal of resources is considered, it is used in a very constrained way [6]. On the other hand, cloud computing elasticity requests powerful acquire operations *as well as* release ones. Let us consider the following problem: given a pool of virtual machine instances and a program that acquires and releases these instances, what is the minimal cardinality of the pool guaranteeing the execution of the program without interruptions caused by lack of virtual machines? A solution to this problem, under the assumption that one can acquire a virtual machine that has been previously released, is useful both for cloud providers and for cloud customers. For the formers, it represents the possibility to estimate *in advance* the resources to allocate to a specific service. For the latter ones, it represents the possibility to pay *exactly* for the resources that are needed.

It is worth to notice that, without a full-fledged release operation, the cost of a concurrent program may be modeled by simply aggregating the sets of operations that can occur in parallel, as in [5]. By full-fledged release operation we mean that it is possible to delegate other (ad-hoc or third party) methods to release resources (by passing them as arguments of invocations). For example, consider the following method

```
Int double_release(Vm x, Vm y) {
  release x; release y;
  return 0 ;
}
```

that takes two machines and simply releases them. The cost of this method depends on the machines in input:

* Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

PPDP '15, July 14–16, 2015, Siena, Italy.
Copyright © 2015 ACM 978-1-4503-3516-4/15/07...$15.00.
http://dx.doi.org/10.1145/2790449.2790524

– it may be `-2` when `x` and `y` are *different* and active;

– it may be `-1` when `x` and `y` are *equal* and active – consider the invocation `double_release(x,x)`;

– it may be `0` when the two machines have been already released.

In this case, one might over-approximate the cost of `double_release` to `0`. However this leads to disregard releases and makes the analysis (too) imprecise.

In order to compute a precise cost of methods like `double_release`, in Section 4 we associate methods with abstract descriptions that carry information about resource usages. These descriptions are called *behavioural types* and are formally connected to the programs by means of a type system.

The analysis of behavioural type is defined in Section 5 by translating them in a code that is adequate for an off-the-shelf solver – the `CoFloCo` solver [11]. As discussed in [7], in order to compute tight upper bounds, we have two functions per method: a function computing the *peak cost* – *i.e.* the worst case cost for the method to complete – and a function computing the *net cost* – *i.e.* the cost of the method after its completion. In fact, the functions that we associate to a method are much more than two. The point is that, if a method has two arguments – see `double_release` – and it is invoked with two *equal* arguments then its cost cannot be computed by a function taking two arguments, but it must be computed by a function with one argument only. This means that, for every method and *every partition of its arguments*, we define two cost functions: one for the peak cost and the other for the net cost. The translation of behavioural types into `CoFloCo` input code has been prototyped and we are therefore able to automatically compute the cost of programs. It is worth to notice that our technique (and, consequently, our prototype) allows us to derive bounds for programs that are better than other techniques, such as those based on amortized analysis. We address this topic in Section 8.

Our technique targets a simple concurrent language with explicit operations of creation and release of resources. The language is defined in Section 2 and we discuss restrictions that ease the development of our technique in Section 3. In Section 6 we outline our correctness proof of the type system with respect to the cost equations. Due to page constraints, the details of the proof are omitted and appear in the full paper. In Section 9 we deliver concluding remarks.

In this paper we use the metaphor of cloud computing and virtual machines. We observe that our technique may be also used for resource analysis of concurrent languages that bear operations of acquire (or creation) and release (such as heaps).

2. The language `vml`

The syntax and the semantics of `vml` are defined in the following two subsections; the third subsection discusses a number of examples.

Syntax. A `vml` program is a sequence of method definitions $T\,\mathtt{m}(\overline{T\ x})\{\,\overline{F\ y\ ;}\ s\,\}$, ranged over by M, plus a main body $\{\overline{F\ z\ ;}\ s'\}$. In `vml` we distinguish between *simple types* T which are either integers `Int` or virtual machines `Vm`, and *types* F, which also include *future types* `Fut<`T`>`. These future types let asynchronous method invocations be typed (see below). The notation $\overline{T\ x}$ denotes any finite sequence of *variable declaration* $T\ x$. The elements of the sequence are separated by commas. When we write $\overline{T\ x\ ;}$ we mean a sequence $T_1\ x_1\ ;\ \cdots\ ;\ T_n\ x_n\ ;$ when the sequence is not empty; we mean the empty sequence otherwise.

The syntax of statements s, expressions with side-effects z and expressions e of `vml` is defined by the following grammar:

$$\begin{array}{rcl}
s & ::= & x = z \ \mid\ \mathtt{if}\ e\,\{\,s\,\}\ \mathtt{else}\ \{\,s\,\} \ \mid\ \mathtt{return}\ e \ \mid\ s\ ;\ s \\
 & \mid & \mathtt{release}\ e \\
z & ::= & e \ \mid\ e!\mathtt{m}(\overline{e}) \ \mid\ e.\mathtt{get} \ \mid\ \mathtt{new\ Vm} \\
e & ::= & \mathtt{this} \ \mid\ se \ \mid\ nse
\end{array}$$

A statement s may be either one of the standard operations of an imperative language plus the `release` x operation which marks the virtual machine x for disposal.

An expression z may change the state of the system. In particular, it may be an *asynchronous* method invocation that does not suspend caller's execution: when the value computed by the invocation is needed then the caller performs a *non blocking* `get` operation: if the value needed by a process is not available then an awaiting process is scheduled and executed. Expressions z also include `new Vm` that creates a new virtual machine. The intended meaning of operations taking place on different virtual machines is that they may execute in parallel, while operations in the same virtual machine interleave their evaluation (even if in the following operational semantics the parallelism is not explicit). The execution of method invocations and creations and releases of machines always returns an erroneous value when executed on a released machine.

A (*pure*) expression e are the reserved identifier `this`, the virtual machines identifiers and the integer expressions. Since our analysis will be parametric with respect to the inputs, we parse integer expressions in a careful way. In particular we split them into *size expressions* se, which are expressions in Presburger arithmetics (this is a decidable fragment of Peano arithmetics that only contains addition), and *non-size expressions* nse, which are the other type of expressions. The syntax of size and non-size expressions is the following:

$$\begin{array}{rcl}
nse & ::= & p \ \mid\ x \ \mid\ nse \leq nse \ \mid\ nse\ \mathtt{and}\ nse \ \mid\ nse\ \mathtt{or}\ nse \\
 & \mid & nse + nse \ \mid\ nse - nse \ \mid\ nse \times nse \ \mid\ nse/nse \\
se & ::= & ve \ \mid\ ve \leq ve \ \mid\ se\ \mathtt{and}\ se \ \mid\ se\ \mathtt{or}\ se \\
ve & ::= & p \ \mid\ x \ \mid\ ve + ve \ \mid\ p \times ve \\
p & ::= & \textit{integer constants}
\end{array}$$

In the whole paper, we assume that sequences of declarations $\overline{T\ x}$ and method declarations $\overline{M}$ do not contain duplicate names. We also assume that `return` statements have no continuation.

Semantics. `vml` semantics is defined as a transition relation between *configurations*, noted *cn* and defined below

$$\begin{array}{rcl}
cn & ::= & \epsilon \ \mid\ \mathit{fut}(f, v) \ \mid\ \mathit{vm}(o, a, p, q) \ \mid\ \mathit{invoc}(o, f, \mathtt{m}, \overline{v}) \ \mid\ cn\ cn \\
p & ::= & \{l \mid \epsilon\} \ \mid\ \{l \mid s\} \\
q & ::= & \epsilon \ \mid\ p \ \mid\ q\, q \\
v & ::= & \textit{integer constants} \ \mid\ o \ \mid\ f \ \mid\ \bot \ \mid\ \top \ \mid\ \mathit{err} \\
l & ::= & [\cdots, x \mapsto v, \cdots]
\end{array}$$

Configurations are sets of elements – therefore we identify configurations that are equal up-to associativity and commutativity – and are denoted by the juxtaposition of the elements *cn cn*; the empty configuration is denoted by ϵ. The transition relation uses two infinite sets of names: *vm names*, ranged over by o, o', $\cdots$ and *future names*, ranged over by f, f', $\cdots$. The function fresh() returns either a fresh vm name or a fresh future name; the context will disambiguate between the twos. We also use l to range over maps from variables to values. The map l also binds the special name destiny to a future value.

Runtime values v are either integers or vm and future names, or two distinct special values denoting a machine alive ($\top$) or dead ($\bot$), or an erroneous value err.

The elements of configurations are

– *virtual machines* $vm(o, a, p, q)$ where o is a vm name; a is either $\top$ or $\bot$ according to the machine is alive or dead, p

is either $\{l \mid \epsilon\}$, representing a terminated statement, or is the *active process* $\{l \mid s\}$, where l returns the values of local variables and s is the continuation; q is a set of processes to evaluate.

– *future binders* $fut(f, v)$. When the value v is $\bot$ then the actual value of f has still to be computed.

– *method invocation messages* $invoc(o, f, \mathtt{m}, \overline{v})$.

The following auxiliary functions are used in the semantic rules (we assume a fixed `vml` program):

– $\mathrm{dom}(l)$ returns the domain of l.

– $l[x \mapsto v]$ is the function such that $(l[x \mapsto v])(x) = v$ and $(l[x \mapsto v])(y) = l(y)$, when $y \neq x$.

– $[\![e]\!]_l$ returns the value of e, possibly retrieving the values of the variables that are stored in l. As regards boolean operations, as usual, `false` is represented by `0` and `true` is represented by a value different from `0`. Operations in `vml` are also defined on the value err: when one of the arguments is err, every operation returns err. $[\![\overline{e}]\!]_l$ returns the tuple of values of $\overline{e}$. When e is a future name, the function $[\![\cdot]\!]_l$ is the identity. Namely $[\![f]\!]_l = f$. It is worth to notice that $[\![e]\!]_l$ is undefined whenever e contains a variable that is not defined in l.

– $\mathrm{bind}(o, f, \mathtt{m}, \overline{v}) = \{[\overline{x} \mapsto \overline{v}, \text{destiny} \mapsto f] \mid s[^o/_{\mathtt{this}}]\}$, where $T\ \mathtt{m}(\overline{T\ x})\{\overline{T'\ z}; s\}$ belongs to the program.

The transition relation rules are collected in Figure 1. They define transitions of virtual machines $vm(o, a, p, q)$ according to the shape of the statement in the active process p. The rules are almost standard, except those about the management of virtual machines and the method invocation, which we are going to discuss.

(NEW-VM) creates a virtual machine and makes it active – rule (NEW-VM). If the virtual machine executing `new Vm` has been already released, then the operation returns an error – rule (NEW-VM-ERR). A virtual machine is disposed by means of the operation `release` x: this amounts to update its state a to $\bot$ – rules (RELEASE-VM) and (RELEASE-VM-SELF). If instead the virtual machine executing the `release` has been already released, then the operation has no effect – rule (RELEASE-BOT).

Rule (ASYNC-CALL) defines asynchronous method invocation $x = e!\mathtt{m}(\overline{e})$. This rule creates a fresh future name that is assigned to the identifier x. The evaluation of the called method is then transferred to the callee virtual machine – rule (BIND-MTD) – and the caller progresses without waiting for callee's termination. If the caller has been already disposed then the invocation returns err – rule (ASYNC-CALL-ERR) The invocation binds err to the future name when either the caller has been released – rule (ASYNC-CALL-ERR) – or the callee machine has been disposed – rule (BIND-MTD-ERR). Rule (READ-FUT) allows the caller to retrieve the value returned by the callee.

The initial configuration of a `vml` program with main function $\{\overline{F\ x\ \ ;}\ s\}$ is

$$vm(start, \top, \{[\text{destiny} \mapsto f_{start}] \mid s\}, \varnothing)$$

where $start$ is a special virtual machine and f_{start} is a fresh future name. As usual, let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$.

Examples. In order to illustrate the features of `vml` we discuss few examples. For every example we also examine the type of output we expect from our cost analysis. We begin with two methods computing the factorial function:

```
Int fact(Int n){
   Fut<Int> x ; Int m ;
   if (n==0) { return 1 ; }
   else { x = this!fact(n-1) ; m = x.get ;
          return m*n ; }
}
Int costly_fact(Int n){
   Fut<Int> x ; Int m ; Vm z ;
   if (n==0) { return 1 ; }
   else { z = new Vm; x = z!fact(n-1) ; m = x.get ;
          release z; return m*n; }
}
```

The method `fact` is the standard definition of factorial with the recursive invocation `fact(n-1)` always performed on the same machine. That is, to compute `fact(n)` one needs one virtual machine. On the contrary, the method `costly_fact` performs the recursive invocation on a new virtual machine `z`. The caller waits for its result, let it be `m`, then it releases the machine `z` and delivers the value `m*n`. Notice that every vm creation occurs before any release operation. As a consequence, `costly_fact` will create as many virtual machines as the argument n. That is, if the application has only k virtual machines then `costly_fact` cab compute factorials up-to $k - 1$ (1 is the virtual machine executing the method).

The analysis of `costly_fact` has been easy because the `release` operation is applied on a locally created virtual machine. Yet, in `vml`, `release` may also apply to method arguments and the presence of this feature in concurrent codes is the major source of difficulties for the analysis. A paradigmatic example is the `double_release` method discussed in Section 1 that may have either a cost of `-2` or of `-1` or of `0`. It is worth to observe that, while over-approximations (*e.g* not counting releases) return (too) imprecise costs, under-approximations may return wrong costs. For example, the following method

```
Int fake_method(Int n) {
   if (n=0) return 0 ;
   else { Vm x ; Fut<Int> f ;
          x = new Vm ; x = new Vm ;
          f = this!double_release(x,x) ; f.get ;
          f = this!fake_method(n-1) ; f.get ;
          return 0 ; }
}
```

creates two virtual machines and releases the second one with `this!double_release(x,x)` before the recursive invocation. We notice that `fake_method(n)` should have cost `n`. However

– an under-approximation of `double_release` (cost `-2`) gives `0` as cost of `fake_method(n)`.

The aim of the following sections is to define a technique for determining the cost of method invocations that makes these costs depend on the identity and on the state of method's arguments, as well as on those arguments that are released.

3. Determinacy of releases of method's arguments

Our cost analysis of virtual machines uses abstract descriptions that carry informations about method invocations and creations and removals of virtual machines. In order to ease the compositional reasonings, method's descriptions also defines the arguments the method releases upon termination. In this contribution we stick to method descriptions that are as simple as possible, namely we assume that the arguments a method releases upon termination are *a set*. In turn, this requires that methods' behaviours are *deterministic* with respect to such releases. To enforce this determinacy, we constrain the language `vml` as follows.

Restriction 1: *the branches in a method body always release the same set of method's arguments.*

For example, methods like

$$\text{(ASSIGN)}\quad \frac{v = \llbracket e \rrbracket_l}{\begin{array}{c} vm(o, a, \{l \mid x = e; s\}, q) \\ \rightarrow vm(o, a, \{l[x \mapsto v] \mid s\}, q) \end{array}}$$

$$\text{(READ-FUT)}\quad \frac{f = \llbracket e \rrbracket_l \quad v \neq \bot}{\begin{array}{c} vm(o, a, \{l \mid x = e\mathtt{.get}; s\}, q)\, \mathit{fut}(f, v) \\ \rightarrow vm(o, a, \{l \mid x = v; s\}, q)\, \mathit{fut}(f, v) \end{array}}$$

$$\text{(ASYNC-CALL)}\quad \frac{o' = \llbracket e \rrbracket_l \quad \overline{v} = \llbracket \overline{e} \rrbracket_l \quad f = \text{fresh}(\,)}{\begin{array}{c} vm(o, \top, \{l \mid x = e!\mathtt{m}(\overline{e}); s\}, q) \\ \rightarrow vm(o, \top, \{l \mid x = f; s\}, q)\ invoc(o', f, \mathtt{m}, \overline{v})\, \mathit{fut}(f, \bot) \end{array}}$$

$$\text{(BIND-MTD)}\quad \frac{\{l \mid s\} = \text{bind}(o, f, \mathtt{m}, \overline{v})}{\begin{array}{c} vm(o, \top, p, q)\ invoc(o, f, \mathtt{m}, \overline{v}) \\ \rightarrow vm(o, \top, p, q \cup \{l \mid s\}) \end{array}}$$

$$\text{(COND-TRUE)}\quad \frac{\llbracket e \rrbracket_l \neq 0 \quad \llbracket e \rrbracket_l \neq err}{\begin{array}{c} vm(o, a, \{l \mid \mathtt{if}\ e\ \mathtt{then}\ \{s_1\}\ \mathtt{else}\ \{s_2\}; s\}, q) \\ \rightarrow vm(o, a, \{l \mid s_1; s\}, q) \end{array}}$$

$$\text{(COND-FALSE)}\quad \frac{\llbracket e \rrbracket_l = 0 \quad \text{or} \quad \llbracket e \rrbracket_l = err}{\begin{array}{c} vm(o, a, \{l \mid \mathtt{if}\ e\ \mathtt{then}\ \{s_1\}\ \mathtt{else}\ \{s_2\}; s\}, q) \\ \rightarrow vm(o, a, \{l \mid s_2; s\}, q) \end{array}}$$

$$\text{(NEW-VM)}\quad \frac{o' = \text{fresh}(\mathtt{VM})}{\begin{array}{c} vm(o, \top, \{l \mid x = \mathtt{new\ Vm}; s\}, q) \\ \rightarrow vm(o, \top, \{l \mid x = o'; s\}, q)\ vm(o', \top, \{\varnothing \mid \varepsilon\}, \varnothing) \end{array}}$$

$$\text{(RELEASE-VM)}\quad \frac{o' = \llbracket e \rrbracket_l \quad o \neq o'}{\begin{array}{c} vm(o, \top, \{l \mid \mathtt{release}\ e; s\}, q)\ vm(o', a', p', q') \\ \rightarrow vm(o, \top, \{l \mid s\}, q)\ \ vm(o', \bot, p', q') \end{array}}$$

$$\text{(RELEASE-VM-SELF)}\quad \frac{o = \llbracket e \rrbracket_l}{\begin{array}{c} vm(o, a, \{l \mid \mathtt{release}\ e; s\}, q) \\ \rightarrow vm(o, \bot, \{l \mid s\}, q) \end{array}}$$

$$\text{(ACTIVATE)}\quad \begin{array}{c} vm(o, a, \{l' \mid \varepsilon\}, q \cup \{l \mid s\}) \\ \rightarrow vm(o, a, \{l \mid s\}, q) \end{array}$$

$$\text{(ACTIVATE-GET)}\quad \frac{f = \llbracket e \rrbracket_{l'}}{\begin{array}{c} vm(o, a, \{l' \mid x = e\mathtt{.get}; s\}, q \cup \{l \mid s\})\, \mathit{fut}(f, \bot) \\ \rightarrow vm(o, a, \{l \mid s\}, q \cup \{l' \mid x = e\mathtt{.get}; s\})\, \mathit{fut}(f, \bot) \end{array}}$$

$$\text{(RETURN)}\quad \frac{v = \llbracket e \rrbracket_l \quad f = l(\text{destiny})}{\begin{array}{c} vm(o, a, \{l \mid \mathtt{return}\ e\}, q)\, \mathit{fut}(f, \bot) \\ \rightarrow vm(o, a, \{l \mid \varepsilon\}, q)\, \mathit{fut}(f, v) \end{array}}$$

$$\text{(NEW-VM-ERR)}\quad \begin{array}{c} vm(o, \bot, \{l \mid x = \mathtt{new\ Vm}; s\}, q) \\ \rightarrow vm(o, \bot, \{l[x \mapsto err]; s\}, q) \end{array}$$

$$\text{(ASYNC-CALL-ERR)}\quad \frac{f = \text{fresh}(\,)}{\begin{array}{c} vm(o, \bot, \{l \mid x = e!\mathtt{m}(\overline{e}); s\}, q) \\ \rightarrow vm(o, \bot, \{l \mid x = f; s\}, q)\, \mathit{fut}(f, err) \end{array}}$$

$$\text{(RELEASE-BOT)}\quad \begin{array}{c} vm(o, \bot, \{l \mid \mathtt{release}\ e; s\}, q) \\ \rightarrow vm(o, \bot, \{l \mid s\}, q) \end{array}$$

$$\text{(BIND-MTD-ERR)}\quad \begin{array}{c} vm(o, \bot, p, q)\ invoc(o, f, \mathtt{m}, \overline{v})\, \mathit{fut}(f, \bot) \\ \rightarrow vm(o, \bot, p, q)\, \mathit{fut}(f, err) \end{array}$$

$$\text{(BIND-PARTIAL)}\quad \begin{array}{c} invoc(err, f, \mathtt{m}, \overline{v})\, \mathit{fut}(f, \bot) \\ \rightarrow \mathit{fut}(f, err) \end{array}$$

$$\text{(CONTEXT)}\quad \frac{cn \rightarrow cn'}{cn\ cn'' \rightarrow cn'\ cn''}$$

Figure 1. Semantics of `vml`.

```
Int foo1(Vm x, Int n) {
    if (n = 0) return 0 ;
    else { release x ; return 0; }
}
```

cannot be handled by our analysis because the else-branch releases the argument `x` while the then-branch does not release anything.

Restriction 2: *method invocations are always synchronized within caller's body.* In this way every effect of a method is computed before its termination. For example, methods like

```
Int foo2(Vm x, Vm y) {
    this!double_release(x,y) ; return 0 ;
}
```

cannot be handled by our analysis because it is not possible to determine that the arguments `x` and `y` of `foo2` will be released or not upon its termination because the invocation to `double_release` is asynchronous.

Restriction 3: *machines that are executing methods that release arguments must be alive.* (This includes the carrier machine, *e.g.* method bodies cannot release the `this` machine.) Here (we are at static time) "alive" means that the machine is either the caller or has been locally created and has not been/being released. For example, in `foo3`

```
Int simple_release(Vm x) { release x; return 0; }
Int foo3(Vm x) {
    Vm z ; Fut<Int> f ;
    z = new Vm ; f = z!simple_release(x) ;
    release z ; f.get ; return 0;
}
```

the machine `z` is released before the synchronisation with the `simple_release` – statement `f.get`. This means that the disposal of `x` depends on scheduler's choice, which means that it is not possible to determine whether `foo3` will release `x` or not. A similar issue arises when the callee of a method releasing arguments is itself an argument. For example, in `foo4`

```
Int foo4(Vm x, Vm y) {
    Fut<Int> f ;
    f = x!simple_release(y) ;
    f.get ; return 0 ;
}
```

it is not possible to determine whether `y` is released or not because the state of `x` cannot be determined.

Restriction 4: *if a method returns a machine, the machine must be new.* For example, consider the following code:

```
Vm identity(Vm x) { return x; }
{
   Vm x ; Vm y ; Vm z ; Fut<Vm> f ; Fut<Int> g ; Int m ;
   x = new Vm ; y = new Vm ;
   f = x!identity(y) ; g = this!simple_release(x);
   z = f.get ; m = g.get ;
   release z ;
}
```

In this case it is not possible to determine whether the value of `z` is `x` or `err` and, therefore, it is not clear whether the cost of `release z` is 0 or `-1`. The problem is `identity`, which returns the argument that is going to be released by a parallel method. The Restriction 4 bans methods like `identity` because it does not return a fresh machine. In fact, such machines cannot be released by a parallel method.

Restrictions 1, 3, and 4 are enforced by the type system in Section 4, in particular by rules (T-METHOD), (T-INVOKE) and (T-RELEASE), and (T-INVOKE) and (T-RETURN), respectively. Restriction 2 is a programming constraint; it may be released by using a continuation passing style that entangles a lot both the type system and the analysis (see [12] for a possible solution that has been designed for deadlock analysis).

4. The behavioural type system of `vml`

Behavioural types are abstract codes highlighting the features of `vml` programs that are relevant for the cost analysis in Section 5. These types support compositional reasonings and are associated to programs by means of a type system that is defined in this section.

The syntax of behavioural types uses *vm names* α, β, γ, $\cdots$, and *future names* f, f', $\cdots$. Sets of vm names will be ranged over by $\mathtt{S}$, $\mathtt{S}'$, $\mathtt{R}$, $\cdots$, and sets of future names will be ranged over by $\mathtt{F}$, $\mathtt{F}'$, $\cdots$. The syntactic rules are presented in Figure 2.

Behavioural types express creations of virtual machines ($\nu\alpha$) and their removal ($\alpha^{\checkmark}$), method invocations ($\nu f : \mathtt{m}\,\alpha(\overline{\mathbb{s}}) \to \mathbb{o}$) and the corresponding retrieval of the value ($f^{\checkmark}$), and the conditionals (respectively $(se)\{\mathbb{c}\} + (\neg se)\{\mathbb{c}'\}$ or $\mathbb{c} + \mathbb{c}'$, according to whether the boolean guard is a size expressions that depends on the arguments of a method or not). We will always shorten the type $\nu f : \mathtt{m}\,\alpha(\overline{\mathbb{s}}) \to \mathbb{o}$ into $\nu f : \mathtt{m}\,\alpha(\overline{\mathbb{s}})$ whenever $\mathbb{o} = \_$.

In order to have a more precise type of continuations, the leaves of behavioural types are labelled with *environments*, ranged over by Γ, Γ', $\cdots$. Environments are maps from method names $\mathtt{m}$ to terms $\alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R}$, from variables to extended values $\mathbb{x}$, from future names to future values, and from vm names to extended values $\mathtt{F}\mathbb{t}$, which are called *vm states* in the following. These environments occurring in the leaves are only used in the typing proofs and are dropped in the final types (method types and the main statement type).

Vm states $\mathtt{F}\mathbb{t}$ are a collection of future names $\mathtt{F}$ plus the value $\mathbb{t}$ of the virtual machines. This $\mathtt{F}$ specifies the set of parallel methods that are going to release the virtual machine; $\mathbb{t}$ defines whether the virtual machine is alive $\top$, or it has been already released ($\bot$) or, according to scheduler's choices, it may be either alive or released (∂). Vm values also include terms α and $\alpha{\downarrow}$. The value α is given to the argument machines of methods (they will be instantiated by the invocations – see the cost analysis in Section 5), the value $\alpha{\downarrow}$ is given to argument values that are returned by methods *and* can be released by parallel methods ($\alpha{\downarrow}$ will be also evaluated in the cost analysis). Vm values are partially ordered by the relation $\leq$ defined by

$$\partial \leq \top \qquad \partial \leq \bot \qquad \alpha{\downarrow} \leq \bot \qquad \alpha{\downarrow} \leq \alpha\ .$$

In the following we will use the partial operation $\mathbb{t} \sqcap \mathbb{t}'$ returning, whenever it exists, the greatest lower bound between $\mathbb{t}$ and $\mathbb{t}'$. For example $\top \sqcap \bot = \partial$, but $\partial \sqcap \alpha{\downarrow}$ is not defined.

The type system uses judgments of the following form:

- $\Gamma \vdash e : \mathbb{x}$ for pure expressions e, $\Gamma \vdash f : \mathbb{z}$ for future names f, and $\Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R}$ for methods.
- $\Gamma \vdash_{\mathtt{S}} z : \mathbb{x}, \mathbb{c} \triangleright \Gamma'$ for expressions with side effects z, where $\mathbb{x}$ is the value, $\mathbb{c}$ is the behavioural type for z and Γ' is the environment Γ *with updates* of variables and future names.
- $\Gamma \vdash_{\mathtt{S}} s : \mathbb{c}$, in this case the updated environments are inside the behavioural type Γ', in correspondence of every branch of its.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$ and the environment update

$$\Gamma[x \mapsto \mathbb{x}](y) \ \overset{def}{=} \ \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

With an abuse of notation (see rule (T-RETURN)), we let $\Gamma[\_ \mapsto \mathbb{x}] \overset{def}{=} \Gamma$ (because $\_$ does not belong to any environment). We will also use the operation and notation below:

– $\mathtt{F}\mathbb{t}{\Downarrow}$ is defined as follows:

$$\mathtt{F}\mathbb{t}{\Downarrow} \ \overset{def}{=} \ \begin{cases} \mathbb{t} & \text{if } \mathtt{F} = \varnothing \\ \partial & \text{if } \mathtt{F} \neq \varnothing \text{ and } \mathbb{t} = \top \\ \bot & \text{if } \mathtt{F} \neq \varnothing \text{ and } \mathbb{t} = \bot \\ \alpha\downarrow & \text{if } \mathtt{F} \neq \varnothing \text{ and } \mathbb{t} = \alpha \end{cases}$$

and we write $(\mathtt{F}_1\mathbb{t}_1, \cdots, \mathtt{F}_n\mathbb{t}_n){\Downarrow}$ for $(\mathtt{F}_1\mathbb{t}_1{\Downarrow}, \cdots, \mathtt{F}_n\mathbb{t}_n{\Downarrow})$.

– the *multihole contexts* $\mathcal{C}[\,]$ defined by the following syntax:

$$\mathcal{C}[\,] \ ::= \ [\,] \ \mid \ \mathbb{a} \,⨟\, \mathcal{C}[\,] \ \mid \ \mathcal{C}[\,] + \mathcal{C}[\,] \ \mid \ (se)\{\mathcal{C}[\,]\}$$

and, whenever $\mathbb{c} = \mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1] \cdots [\mathbb{a}_n \triangleright \Gamma_n]$, then $\mathbb{c}[x \mapsto \mathbb{x}]$ is defined as $\mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1[x \mapsto \mathbb{x}]] \cdots [\mathbb{a}_n \triangleright \Gamma_n[x \mapsto \mathbb{x}]]$.

The type system for expressions is reported in Figure 3. It is worth to notice that this type system is not standard because (size) expressions containing method's arguments are typed with the expressions themselves. This is crucial in the cost analysis of Section 5.

The type system for expressions with side effects and statements is reported in Figure 4. We discuss rules (T-INVOKE), (T-GET), (T-RELEASE), and (T-NEW).

Rule (T-INVOKE) types method invocations $e!\mathtt{m}(\overline{e})$ by using a fresh future name f that is associated to the method name, the vm name of the callee and the arguments. The relevant point is the value of f in the updated environment. This value contains the returned value, the vm name of the callee and its state, and the set of the arguments that the method is going to remove. The vm state of the callee will be used when the method is synchronized to update the state of the returned object, if any (see rule (T-GET)). It is important to observe that the environment returned by (T-INVOKE) is updated with information about vm names released by the method: every such name will contain f in its state. Next we discuss the constraints in the second line and third line of the premise of (T-INVOKE). Assuming that the callee has not been already released ($\Gamma(\alpha) \neq \mathtt{F}\bot$), there are two cases:

(*i*) either $\Gamma(\alpha) = \varnothing\top$ or α is the caller object α': namely the callee is alive because it has been created by the caller or it is the caller itself,

(*ii*) or $\Gamma(\alpha) \neq \varnothing\top$: this case has two subcases, namely either (*ii.a*) the callee is being released by a parallel method or (*ii.b*) it is an argument of the caller method – see rule (T-METHOD).

While in (*i*) we admit that the invoked method releases vm names, in case (*ii*) we forbid any release, as we discussed in Restriction 3 in Section 3. We observe that, in case (*ii.b*), being α an argument of the method, it may retain any state when the method is invoked and, for reasons similar to (*ii.a*), it is not possible to determine at static time the exact subset of $\mathtt{R}$ that will be released. This constraint enforces Restriction 3 in Section 3. The constraint in the third line of the premise of (T-INVOKE) enforces Restriction 3 to the other invocations in parallel and to the object executing $e!\mathtt{m}(\overline{e})$.

Rule (T-GET) defines the synchronisation with a method invocation that corresponds to a future f. Let $(\mathbb{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R})$ be the value of f in the environment. Since $\mathtt{R}$ defines the resources of the caller that are released, we record in the returned environment Γ' that these resources are no more available. Γ' also records the state of the returned vm name. If the returned value is a virtual machine that has been created by the method of f, its state is the same of the callee vm name (which may have been updated since the invocation), namely the value of $\mathbb{t} \sqcap (\Gamma(\alpha){\Downarrow})$.

$$\begin{array}{rcll}
\mathbb{o} & ::= & \_ \mid \alpha & \text{basic value}\\
\mathbb{t} & ::= & \alpha \mid \alpha{\downarrow} \mid \partial \mid \bot \mid \top & \text{vm value}\\
se & ::= & \textit{integer constant} \mid x \mid (\mathbb{t} \leq \bot) \mid (\mathbb{t} \leq \top) \mid se\ \mathtt{op}'\ se & \text{size expression}\\
\mathtt{op}' & ::= & + \mid - \mid = \mid \leq \mid \geq \mid \wedge \mid \vee & \text{linear operation}\\
\mathbb{r}, \mathbb{s} & ::= & \mathbb{o} \mid se & \text{typing value}\\
\mathbb{z} & ::= & (\mathbb{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R}) \mid \mathbb{o} & \text{future value}\\
\mathbb{x} & ::= & \_ \mid \mathtt{F}\mathbb{t} \mid f \mid \mathbb{z} & \text{extended value}\\
\mathbb{a} & ::= & \mathtt{0} \mid \nu\alpha \mid \nu f : \mathtt{m}\,\alpha(\overline{\mathbb{s}}) \to \mathbb{o} \mid \alpha^{\checkmark} \mid f^{\checkmark} & \text{atom}\\
\mathbb{c} & ::= & \mathbb{a} \triangleright \Gamma \mid \mathbb{a} \fatsemi \mathbb{c} \mid \mathbb{c} + \mathbb{c} \mid (se)\{\mathbb{c}\} & \text{behavioural type}
\end{array}$$

Figure 2. Behavioural Types Syntax

(T-VAR)
$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

(T-PRIMITIVE)
$$\Gamma \vdash \mathtt{p} : \mathtt{p}$$

(T-OP)
$$\frac{\Gamma \vdash e_1 : se_1 \qquad \Gamma \vdash e_2 : se_2}{\Gamma \vdash e_1\ \mathtt{op}'\ e_2 : se_1\ \mathtt{op}'\ se_2}$$

T-UNIT
$$\frac{\Gamma \vdash e : se}{\Gamma \vdash e : \_}$$

(T-OP-UNIT)
$$\frac{\Gamma \vdash e_1 : \_ \quad \mathtt{or} \quad \Gamma \vdash e_2 : \_ \quad \mathtt{or} \quad \mathtt{op} \in \{*, /\}}{\Gamma \vdash e_1\ \mathtt{op}\ e_2 : \_}$$

(T-PURE)
$$\frac{\Gamma \vdash e : \mathbb{x}}{\Gamma \vdash e : \mathbb{x}, \mathtt{0} \triangleright \Gamma}$$

(T-METHOD-SIG)
$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R} \quad \overline{\beta} \subseteq \mathit{fv}(\alpha, \overline{\mathbb{r}}, \mathbb{o})\\ \sigma \text{ is a vm renaming such that } \mathbb{o} \notin \mathit{fv}(\alpha, \overline{\mathbb{r}}) \text{ implies } \sigma(\mathbb{o}) \text{ fresh}\end{array}}{\Gamma \vdash \mathtt{m}\,\sigma(\alpha)(\sigma(\overline{\mathbb{r}})) : \sigma(\mathbb{o}), \sigma(\mathtt{R})}$$

Figure 3. Typing rules for expressions

(T-ASSIGN-VAR)
$$\frac{\Gamma(x) = \mathbb{x} \qquad \Gamma \vdash_{\mathtt{S}} z : \mathbb{x}', \mathbb{c}}{\Gamma \vdash_{\mathtt{S}} x = z : \mathbb{c}[x \mapsto \mathbb{x}']}$$

(T-INVOKE)
$$\frac{\begin{array}{c}\Gamma \vdash e : \alpha \qquad \Gamma \vdash \overline{e} : \overline{\mathbb{s}} \qquad \Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{s}}) : \mathbb{o}, \mathtt{R} \qquad \Gamma \vdash \mathtt{this} : \alpha'\\ \Gamma(\alpha) \neq \mathtt{F}\bot \quad \text{and} \quad ((\Gamma(\alpha) \neq \varnothing\top \text{ and } \alpha \neq \alpha') \quad \text{implies} \quad \mathtt{R} = \varnothing)\\ \mathtt{R} \cap \Big(\{\alpha'\} \cup \{\beta \mid f' \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(f') = (\mathbb{o}', \beta, \mathbb{x}, \mathtt{R}') \text{ and } \mathtt{R}' \neq \varnothing\}\Big) = \varnothing\\ f \text{ fresh} \qquad \Gamma' = \Gamma[\beta \mapsto (\{f\} \cup \mathtt{F}')\mathbb{t}]^{\beta \in \mathtt{R}, \Gamma(\beta) = \mathtt{F}'\mathbb{t}}\end{array}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \nu f\!: \mathtt{m}\,\alpha(\overline{\mathbb{s}}) \to \mathbb{o} \triangleright \Gamma'[f \mapsto (\mathbb{o}, \alpha, \Gamma(\alpha), \mathtt{R})]}$$

(T-INVOKE-BOT)
$$\frac{\Gamma \vdash e : \alpha \qquad \Gamma(\alpha) = \mathtt{F}\bot \qquad f \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \mathtt{0} \triangleright \Gamma'[f \mapsto \_]}$$

(T-GET)
$$\frac{\begin{array}{c}\Gamma \vdash x : f \qquad \Gamma \vdash f : (\mathbb{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R})\\ \mathtt{R}' = \mathit{fv}(\mathbb{o}) \setminus \mathtt{R} \qquad \mathbb{t}' = \mathbb{t} \sqcap (\Gamma(\alpha){\Downarrow})\\ \Gamma' = \Gamma[\beta \mapsto \varnothing\bot]^{\beta \in \mathtt{R}}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}'}\end{array}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, f^{\checkmark} \triangleright \Gamma'[f \mapsto \mathbb{o}]}$$

(T-GET-DONE)
$$\frac{\Gamma \vdash x : f \qquad \Gamma \vdash f : \mathbb{o}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, \mathtt{0} \triangleright \Gamma}$$

(T-NEW)
$$\frac{\beta \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} \mathtt{new}\ \mathtt{Vm} : \beta, \nu\beta \triangleright \Gamma[\beta \mapsto \varnothing\top]}$$

(T-RELEASE)
$$\frac{\begin{array}{c}\Gamma \vdash x : \alpha\\ \alpha \notin \{\beta \mid f' \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(f') = (\mathbb{o}', \beta, \mathbb{x}, \mathtt{R}') \text{ and } \mathtt{R}' \neq \varnothing\}\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{release}\ x : \alpha^{\checkmark} \triangleright \Gamma[\alpha \mapsto \varnothing\bot]}$$

(T-IF)
$$\frac{\Gamma \vdash e : se \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathbb{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathbb{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\,\{\, s_1 \,\}\ \mathtt{else}\,\{\, s_2 \,\} : (se)\{\, \mathbb{c}_1 \,\} + (\neg se)\{\, \mathbb{c}_2 \,\}}$$

(T-IF-ND)
$$\frac{\Gamma \vdash e : \_ \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathbb{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathbb{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\,\{\, s_1 \,\}\ \mathtt{else}\,\{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2}$$

(T-SEQ)
$$\frac{\Gamma \vdash_{\mathtt{S}} s_1 : \mathcal{C}[\mathbb{a}_1 \triangleright \Gamma_1] \cdots [\mathbb{a}_n \triangleright \Gamma_n] \qquad (\Gamma_i \vdash_{\mathtt{S}} s_2 : \mathbb{c}'_i)^{i \in 1..n}}{\Gamma \vdash_{\mathtt{S}} s_1; s_2 : \mathcal{C}[\mathbb{a}_1 \fatsemi \mathbb{c}'_1] \cdots [\mathbb{a}_n \fatsemi \mathbb{c}'_n]}$$

(T-RETURN)
$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbb{o} \qquad \Gamma \vdash \mathtt{destiny} : \mathbb{o}'\\ \mathbb{o} \notin \mathtt{S}\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{return}\ e : \mathtt{0} \triangleright \Gamma[\mathbb{o}' \mapsto \Gamma(\mathbb{o})]}$$

Figure 4. Type rules for expressions with side effects and statements.

Rule (T-RELEASE) models the removal of a vm name α. The premise in the second line verifies that the disposal do not address machines that are executing methods, as discussed in Restriction 3 of Section 3.

The type system of `vml` is completed with the rules for method declarations and programs, given in Figure 5.

Without loss of generality, rule (T-METHOD) assumes that formal parameters of methods are ordered: those of `Int` type occur before those of `Vm` type. We observe that the environment typing the method body binds integer parameters to their same name, while the other ones are bound to fresh vm names (this lets us to have a more precise cost analysis in Section 5). We also observe that the returned value $\mathbb{o}$ may be either $\_$ or a fresh vm name ($\mathbb{o} \notin \{\alpha\} \cup \overline{\beta}$) as discussed in Restriction 4 of Section 3. The constraints in the third line of the premises of (T-METHOD) implement Restriction 1 of Section 3. We also observe that $(\Gamma_i(\gamma) = \Gamma_j(\gamma))^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup \mathit{fv}(\mathbb{o})}$ guarantees that every branch of the be-

(T-METHOD)

$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{x},\overline{\beta}) : \mathbb{o}, \mathtt{R} \qquad \mathtt{S} = \{\alpha\} \cup \overline{\beta} \qquad \mathbb{o} \notin \mathtt{S} \\ \Gamma[\mathtt{this} \mapsto \alpha][\mathtt{destiny} \mapsto \mathbb{o}][\overline{x} \mapsto \overline{x}][\overline{z} \mapsto \overline{\beta}][\alpha \mapsto \varnothing\alpha][\overline{\beta} \mapsto \varnothing\overline{\beta}] \vdash_{\mathtt{S}} s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n] \\ \Big(\Gamma_i(\gamma) = \Gamma_j(\gamma)\Big)^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup fv(\mathbb{o})} \qquad \mathtt{R} = (\mathtt{S} \cup fv(\mathbb{o})) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\}\end{array}}{\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{Int}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;\ s}\}\ : \mathtt{m}\ \alpha(\overline{x},\overline{\beta})\ \{\,\mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing]\,\} : \mathbb{o},\ \mathtt{R}}$$

(T-PROGRAM)

$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}} \qquad \Gamma \vdash_{\text{start}} s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]}{\Gamma \vdash \overline{M}\ \{\overline{F\ x\ ;\ s}\} : \overline{\mathbb{C}},\ \mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing]}$$

Figure 5. Behavioural typing rules of method and programs.

havioural type creates a new vm name and, by rule (T-RETURN), the state of the chosen vm name must be always the same.

We display behavioural types examples by using codes from Sections 1 and 2. Actually, the following types do not abstract a lot from codes because the programs of the previous sections have been designed for highlighting the issues of our technique.

The behavioural types of `fact` and `costly_fact` are the following ones

```
fact $\alpha$(n) {
 (n==0){ 0 }
 +(n>0){ $\nu\, y$:fact $\alpha(n-1)$ $\fatsemi$ $y^{\checkmark}$ }
} - , { }
```

```
costly_fact $\alpha$(n) {
 (n==0){ 0 }
 +(n>0){ $\nu\beta$ $\fatsemi$
         $\nu\, x$ : costly_fact $\beta(n-1)$ $\fatsemi$
         $x^{\checkmark}$ $\fatsemi$ $\beta^{\checkmark}$ }
} - , { }
```

and it is worth to highlight that the type of `costly_fact` records the order between the recursive invocation and the release of the machine.

The behavioural type of `double_release` is the following one

```
double_release $\alpha$($\beta$, $\gamma$) {$\beta^{\checkmark}$ $\fatsemi$ $\gamma^{\checkmark}$ } - , {$\beta$, $\gamma$}
```

It is worth to notice that the releases $\beta^{\checkmark}$ and $\gamma^{\checkmark}$ in `double_release` are conditioned by the values of β and γ when the method is invoked.

5. The analysis of behavioural types

The types returned by the system in Section 4 are used to compute the resource cost of a `vml` program. This computation is performed by an off-the-shelf solver – the `CoFloCo` solver [11] – that takes in input a set of so-called *cost equations*. `CoFloCo` cost equations are terms

$$m(\overline{x}) = \mathit{exp} \quad [\mathit{se}]$$

where m is a (cost) function symbol, *exp* is an expression that may contain (cost) function symbols applications (we do not define the syntax of *exp*, which may be derived by the following equations; the reader is referred to [11]), and *se* is a size expression whose variables are contained in $\overline{x}$.

Basically, our translation maps method types into cost equations, where

- method invocations are translated into function applications,
- virtual machine creations are translated into a `+1` cost,
- virtual machine releases are translated into a `-1` cost,

There are two function calls for every method invocation: one returns the maximal number of resources needed to execute a method `m`, called *peak cost* of `m` and noted $\mathtt{m}_{\text{peak}}$, and the other returns the number of resources the method `m` creates without releasing, called *net cost* of `m` and noted $\mathtt{m}_{\text{net}}$. These functions are used to define the cost of sequential execution and parallel execution of methods. For example, omitting arguments of methods, the cost of the sequential composition of two methods `m` and $\mathtt{m}'$ is the maximal value between $\mathtt{m}_{\text{peak}}$, $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$, and $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{net}}$; while the cost of the parallel execution of `m` and $\mathtt{m}'$ is the maximal value between $\mathtt{m}_{\text{peak}} + \mathtt{m}'_{\text{peak}}$, $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$, $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$, and $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{net}}$.

There are two difficulties that entangle our translation, both related to method invocations: the management of arguments' identities and of arguments' values.

Arguments' identities. Consider the code

```
Int simple_release(Vm x) { release x ; return 0 ; }

Int m(Vm x, Vm y) {
    Fut<Int> f; f = this!simple_release(x); release y; f.get;
    return 0;
}
```

The behavioural types of these methods are

```
simple_release $\alpha(\beta)$\{ $\beta^{\checkmark}$ \} _, \{$\beta$\}
m $\alpha(\beta,\gamma)$\{$\nu\, f$ : simple_release $\alpha(\beta)$ $\fatsemi$ $\gamma^{\checkmark}$ $\fatsemi$ $f^{\checkmark}$\} _, \{$\beta,\gamma$\}
```

We notice that, in the type of `m`, there is not enough information to determine whether $\gamma^{\checkmark}$ will have a cost equal to `-1` or `0`. In fact, while in typing rules of methods the arguments are assumed to be pairwise different – see rule (T-METHOD) –, it is not the case for invocations. For instance, if `m` is invoked with two arguments that are equal – $\beta = \gamma$ – then γ is going to be released by the invocation $\mathtt{free}(\beta)$ and therefore it counts `0`. To solve this problem of arguments' identity, we refine even more the translation of a method type, which now depends on an equivalence relation telling which of the vm names in parameter are actually equal or not. Hence, the above method `m` is translated in four cost functions: $\mathtt{m}_{\text{peak}}^{\{1\},\{2\}}(x,y)$ and $\mathtt{m}_{\text{net}}^{\{1\},\{2\}}(x,y)$, which correspond to the invocations where $x \neq y$, and $\mathtt{m}_{\text{peak}}^{\{1,2\}}(x)$ and $\mathtt{m}_{\text{net}}^{\{1,2\}}(x)$, which correspond to the invocations where $x = y$. (The equivalence relation in the superscript never mention `this`, which is also an argument, because, in this case `this` cannot be identified with the other arguments, see below.)

The following function `EqRel` computes the equivalence relation corresponding to a specific method call; `EqRel` takes a tuple of vm names and returns an equivalence relation on indices of the tuple:

$$\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n) \;=\; \bigcup_{i \in 0..n} \{\ \{j \mid \alpha_j = \alpha_i\}\ \}$$

Let $\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n)(\beta_0, \cdots, \beta_n)$ be the tuple $(\beta_{i_1}, \cdots, \beta_{i_k})$, where $i_1, \cdots, i_k$ are *canonical representatives* of the sets in

$\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n)$ (we take the vm name with the least index in every set). We observe that, by definition, $\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n)(\alpha_0, \cdots, \alpha_n)$ is a tuple of pairwise different vm names (in $\alpha_0, \cdots, \alpha_n$).

Without loosing in generality, we will always assume that the canonical representative of a set containing 0 is always 0. This index represents the `this` object and we remind that, by Restriction 3 in Section 3, such an object cannot be released. This is the reason why, in the foregoing discussion about the method `m`, we did not mentioned `this`. Additionally, in order to simplify the translation of method invocations, we also assume that the argument `this` is always different from other arguments (the general case just requires more details).

(Re)computing argument's states. In Section 4 we computed the state of every machine in order to enforce the restrictions in Section 3. In this section we mostly compute them again for a different reason: obtaining a (more) precise cost analysis. Of course one might record the computation of vm states in behavioural types. However, this solution has the drawback that behavioural types become unintelligible because they carry informations that are needed by the analyser.

Let a *translation environment*, ranged over Ψ, Ψ', be a mapping from vm names to vm states and from future names to triples $(\Psi', \mathtt{R}, \mathtt{m}\ \beta(\overline{se}, \overline{\beta}) \to \mathbb{o})$, where Ψ' is a translation environment defined on vm names only, called *vm-translation environment*. We define the following auxiliary functions

– let Ψ be a vm-translation environment. Then

$$\Psi|_X(\alpha) \overset{def}{=} \begin{cases} \Psi(\alpha) & \text{if } \alpha \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

– the *update of a vm-translation environment* Ψ with respect to f and Ψ', written $\Psi \searrow^{f} \Psi'$, returns a vm-translation environment defined as follows:

$$(\Psi \searrow^{f} \Psi')(\alpha) \overset{def}{=} \begin{cases} (\mathtt{F}' \setminus \{f\})(\mathbb{t} \sqcap \mathbb{t}') & \text{if } \Psi(\alpha) = \mathtt{F}\mathbb{t} \\ & \text{and } \Psi'(\alpha) = \mathtt{F}'\mathbb{t}' \\ \\ \text{undefined} & \text{otherwise} \end{cases}$$

This operation $\Psi \searrow^{f} \Psi'$ updates the vm-translation environment Ψ of a method invocation, which is stored in the future f, with respect to the translation environment at the synchronisation point. It is worth to observe that, by definition of our type system and the following translation function, the values of $\Psi(\alpha)$ and $\Psi'(\alpha)$ are related. In particular, if $\mathbb{t} = \alpha$ then $\mathbb{t}'$ can be either α or $\alpha{\downarrow}$ (the machine is released by a method that has been invoked in parallel) or $\bot$ (the machine has been released before the `get` operation on the future f); if $\mathbb{t} = \top$ then $\mathbb{t}'$ can be either $\top$ or ∂ (the machine is released by a method that has been invoked in parallel) or $\bot$ (the machine has been released before the `get` operation).

– the *merge operation*, noted $\Psi(\Delta)$, where Ψ is a vm-translation environment and Δ is an equivalence relation, returns a *substitution* defined as follows. Let

$$\mathbb{t} \otimes^{\alpha} \mathbb{t}' \overset{def}{=} \begin{cases} \bot & \text{if } \mathbb{t} = \bot \text{ or } \mathbb{t}' = \bot \\ \alpha & \text{if } \mathbb{t} \text{ and } \mathbb{t}' \text{ are variables} \\ \alpha{\downarrow} & \text{otherwise} \end{cases}$$

$$\mathtt{F}_1\mathbb{t}_1 \otimes^{\alpha} \mathtt{F}_2\mathbb{t}_2 \overset{def}{=} (\mathtt{F}_1 \cup \mathtt{F}_2)(\mathbb{t}_1 \otimes^{\alpha} \mathbb{t}_2)$$

Then

$$\Psi(\Delta) \ : \ \alpha \mapsto \bigotimes{}^{\Delta(\alpha)}\{\Psi(\beta) \mid \beta \in \mathrm{dom}(\Psi) \text{ and } \Delta(\beta) = \Delta(\alpha)\}$$

for every $\alpha \in \mathrm{dom}(\Psi)$.

The operator $\otimes^{\alpha}$ has not been defined on vm values as ∂ or $\top$ because we merge vm names whose image by Ψ can be either $\mathtt{F}\beta$ or $\mathtt{F}\beta{\downarrow}$ or $\mathtt{F}\bot$. As a notational remark, we observe that $\Psi(\Delta)$ is noted as a map $[\alpha_1 \mapsto \mathtt{F}_1\mathbb{t}_1, \cdots, \alpha_n \mapsto \mathtt{F}_n\mathbb{t}_n]$ instead of the standard notation $[^{\mathtt{F}_1\mathbb{t}_1, \cdots, \mathtt{F}_n\mathbb{t}_n}/_{\alpha_1, \cdots, \alpha_n}]$. These two notations are clearly equivalent: we prefer the former one because it will let us to write $\Psi(\Delta)(\alpha)$ or even $\Psi(\Delta)(\alpha_1, \cdots, \alpha_n)$ with the obvious meanings.

To clarify the reason for a merge operator, consider the atom $f^{\checkmark}$ within a behavioural type that binds f to $\mathtt{foo}\,\alpha(\beta, \gamma)$. Assume to evaluate this type with $\Delta = \{\beta, \gamma\}$. That is, the two arguments are actually identical. What are the values of β and γ for evaluating $\mathtt{foo}^{\Delta}_{\text{peak}}$ and $\mathtt{foo}^{\Delta}_{\text{net}}$? Well, we have

1. to select the representative between β and γ: it will be $\Delta(\beta)$ (which is equal to $\Delta(\gamma)$;
2. to take a value that is smaller than $\Psi(\beta)$ and $\Psi(\gamma)$ (but greater than any other value that is smaller);
3. to substitute β and γ with the result of 2.

For instance, let $\Psi = [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta{\downarrow}, \gamma \mapsto \varnothing\gamma]$ and $\Delta(\beta) = \Delta(\gamma) = \beta$. We expect that a value for the item 2 above is $\varnothing\beta{\downarrow}$ and the substitution of the item 3 is $[^{\varnothing\beta{\downarrow}, \varnothing\beta{\downarrow}}/_{\beta, \gamma}]$. Formally, the operation returning the value for 2 is $\otimes^{\beta}$ and the the substitution of item 3 is the output of the merge operation.

The translation function. The translation function, called `translate`, is structured in three parts that respectively correspond to simple atoms, full behavioural types, and method types and full programs. `translate` carries five arguments:

1. Δ is the *equivalence relation on formal parameters* identifying those that are equal. We assume that $\Delta(x)$ returns the unique representative of the equivalence class of x. Therefore we can use Δ also as a substitution operation.
2. Ψ is the *translation environment* which stores temporary information about futures that are active (unsynchronised) and about the state of vm names;
3. α is the name of the virtual machine of the current behavioural type;
4. $\overline{e}$ is the sequence of (over-approximated) costs of the current execution branch;
5. the behavioural type being translated; it may be either $\mathbb{a}$, $\mathbb{c}$ or $\overline{\mathbb{C}}$.

In the definition of `translate` we use the two functions

$$\mathtt{CNEW}(\alpha) = \begin{cases} \mathtt{0} & \alpha = \bot \\ \mathtt{1} & \text{otherwise} \end{cases} \qquad \mathtt{CREL}(\alpha) = \begin{cases} \mathtt{-1} & \alpha = \top \\ \mathtt{0} & \text{otherwise} \end{cases}$$

The left-hand side function is used when a virtual machine is created. It returns `1` or `0` according to the virtual machine that is executing the code can be alive ($\alpha \neq \bot$) or not, respectively. The right-hand side function is used when a virtual machine is released (in correspondence of atoms $\beta^{\checkmark}$). The release is effectively computed – value `-1` – only when the virtual machine that is executing the code is alive ($\alpha = \top$).

Finally, we will assume the presence of a lookup function `lookup` that takes method invocations $\mathtt{m}\ \alpha(\overline{\mathbb{r}}, \overline{\beta})$ and returns tuples $\mathbb{c} : \mathbb{o}$, $\mathtt{R}$. This function is left unspecified.

The definition of `translate` follows. We begin with the translation of atoms.

$$
\mathtt{translate}[\Delta, \Psi, \alpha](\overline{e}; e)(\mathbb{a}) \; = \;
\begin{cases}
(\Psi, \overline{e}; e) \\
\qquad \text{when } \mathbb{a} = \mathtt{0} \\[4pt]
(\Psi[\beta \mapsto \varnothing\top], \overline{e}; e; e + \mathtt{CNEW}(\mathbb{t})) \\
\qquad \text{when } \mathbb{a} = \nu\beta \quad \text{and} \quad \Psi(\alpha) = \mathtt{F}\mathbb{t} \\[4pt]
(\Psi[\Delta(\beta) \mapsto \varnothing\bot], \overline{e}; e; e + \mathtt{CREL}(\mathbb{t})) \\
\qquad \text{when } \mathbb{a} = \beta^{\checkmark} \quad \text{and} \quad \Psi(\Delta(\beta)) = \mathtt{F}\mathbb{t} \\[4pt]
(\Psi'[f \mapsto (\Psi|_{\Delta(\beta,\overline{\beta})}, \mathtt{R}, \mathtt{m}\, \Delta(\beta)(\overline{\mathbb{r}}, \Delta(\overline{\beta}) \to \mathbb{o}))], \overline{e}; e; e + f) \\
\qquad \text{when } \mathbb{a} = \nu f : \mathtt{m}\, \beta(\overline{\mathbb{r}}, \overline{\beta}) \to \mathbb{o} \\
\qquad \text{and } \Psi' = \Psi[\beta \mapsto (\mathtt{F} \cup \{f\})\mathbb{t}]^{\beta \in \mathtt{R}, \Psi(\beta) = \mathtt{F}\mathbb{t}} \\
\qquad \text{and } \mathtt{lookup}(\mathtt{m}\, \Delta(\beta)(\overline{se}, \Delta(\overline{\beta}))) = \mathbb{c} : \mathbb{o},\ \mathtt{R} \\[4pt]
(\Psi'' \setminus f,\ (\overline{e}; e)\sigma; (e)\sigma' + \sum_{\gamma \in \Delta(\mathtt{R}), \Theta(\gamma) = \mathtt{F}\mathbb{t}, \mathtt{F} \neq \varnothing} \mathtt{CREL}(\mathbb{t}) \\
\qquad \text{when } \mathbb{a} = f^{\checkmark} \quad \text{and} \quad \Psi(f) = (\Psi', \mathtt{R}, \mathtt{m}\, \beta(\overline{\mathbb{r}}, \overline{\beta}) \to \mathbb{o}) \\
\qquad \text{and } \mathtt{EqRel}(\beta, \overline{\beta}) = \Xi \quad \text{and} \quad \Theta = \Psi' \overset{f}{\searrow} \Psi \\
\qquad \text{and } \sigma = [\mathtt{m}^{\Xi}_{\text{peak}}(\overline{\mathbb{r}}, \Theta(\Xi(\beta, \overline{\beta})) \Downarrow)/_{f}] \\
\qquad \text{and } \sigma' = [\mathtt{m}^{\Xi}_{\text{net}}(\overline{\mathbb{r}}, \Theta(\Xi(\beta, \overline{\beta})) \Downarrow)/_{f}] \\
\qquad \text{and } \Psi'' = \Psi[\gamma \mapsto \varnothing\bot]^{\gamma \in \mathtt{R}}[\gamma' \mapsto \Theta(\beta)]^{\gamma' \in \mathit{fv}(\mathbb{o}) \setminus \mathtt{R}}
\end{cases}
$$

In the definition of `translate` we always highlight the last expression in the sequence of costs of the current execution branch (the fourth input). This is because the cost of the parsed atom applies to it, except for the case of $f^{\checkmark}$. In this last case, let $\overline{e}; e$ be the expression. Since the atom expresses the synchronisation of f, $\overline{e}; e$ will have occurrences of f. In this case, the function `translate` has to compute two values: the maximum number of resources used by (the method corresponding to) f during its execution – the *peak cost* used in the substitution σ – and the resources used upon the termination of (the method corresponding to) f – the *net cost* used in the substitution σ'. In particular, this last value has to be decreased by the number of the resources released by the method. This is the purpose of the addend $\sum_{\gamma \in \Delta(\mathtt{R}), \Theta(\gamma) = \mathtt{F}\mathbb{t}, \mathtt{F} \neq \varnothing} \mathtt{CREL}(\mathbb{t})$ that remove machines that are going to be removed by parallel methods (the constraint $\mathtt{F} \neq \varnothing$) because the other ones have been already counted both in the peak cost and in the net cost. We observe that the instances of the method $\mathtt{m}_{\text{peak}}$ and $\mathtt{m}_{\text{net}}$ that are invoked are those corresponding to the equivalence relation of the tuple $(\beta, \overline{\beta})$.

The of behavioural types is given by composing the definitions of the atoms. In this case, the output of `translate` is a set of cost equations.

$$
\mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se})\{\, \overline{e} \,\}, \mathbb{c}) \; = \;
\begin{cases}
\{(\mathit{se}')\{\, \overline{e'} \,\}\} \quad \text{when } \mathbb{c} = \mathbb{a} \rhd \varnothing \\
\qquad \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se})\{\, \overline{e} \,\}, \mathbb{a}) = (\Psi', (\mathit{se}')\{\, \overline{e'} \,\}) \\[4pt]
C'' \qquad \text{when } \mathbb{c} = \mathbb{a} \mathbin{;} \mathbb{c}' \\
\qquad \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se})\{\, \overline{e} \,\}, \mathbb{a}) = (\Psi', \{(\mathit{se}')\{\, \overline{e'} \,\}\}) \\
\qquad \text{and } \mathrm{dom}(\Psi') \setminus \mathrm{dom}(\Psi) = \mathtt{S} \\
\qquad \text{and } \mathtt{translate}(\Delta \cup \{\mathtt{S}\}, \Psi', \alpha, (\mathit{se}')\{\, \overline{e'} \,\}, \mathbb{c}') = (\Psi'', C'') \\[4pt]
C' \cup C'' \quad \text{when } \mathbb{c} = \mathbb{c}_1 + \mathbb{c}_2 \\
\qquad \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se})\{\, \overline{e} \,\}, \mathbb{c}_1) = (\Psi', C') \\
\qquad \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se})\{\, \overline{e} \,\}, \mathbb{c}_2) = (\Psi'', C'') \\[4pt]
C' \qquad \text{when } \mathbb{c} = (\mathit{se}')\{\mathbb{c}'\} \\
\qquad \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se} \wedge \mathit{se}')\{\, \overline{e} \,\}, \mathbb{c}') = (\Psi', C') \\[4pt]
C' \qquad \text{when } \mathbb{c} = (e')\{\mathbb{c}'\} \text{ and } e' \text{ contains } \_ \\
\qquad \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (\mathit{se})\{\, \overline{e} \,\}, \mathbb{c}') = (\Psi', C')
\end{cases}
$$

The translation of method types and behavioural type programs is given below. Let $\mathcal{P}$ be the set of partitions of $1..n$. Then

$$
\mathtt{translate}(\mathtt{m}\, \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_n)\ \{\, \mathbb{c} \,\} : \mathbb{o},\ \mathtt{R}) \quad = \quad \bigcup_{\Xi \in \mathcal{P}} \mathtt{translate}(\Xi, \mathtt{m}\, \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_n)\ \{\, \mathbb{c} \,\} : \mathbb{o},\ \mathtt{R})
$$

where $\mathtt{translate}(\Xi, \mathtt{m}\, \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_n)\ \{\, \mathbb{c} \,\} : \mathbb{o},\ \mathtt{R})$ is defined as follows. Let

$$\Delta = \{\, \{\, \alpha_{i_1}, \ldots, \alpha_{i_m} \,\} \mid \{\, i_1, \ldots, i_m \,\} \in \Xi \,\}$$

$$[\![\Delta]\!] = \{\alpha \mapsto \varnothing\alpha \mid \alpha = \Delta(\alpha)\}$$

$$\mathtt{translate}(\Delta,\ [\![\Delta]\!], \alpha_1)(0)(\mathbb{c}) = \bigcup_{i=1}^{n} (\mathit{se}_i)\{\, e_{1,i}; \ldots; e_{h_i,i} \,\}$$

Then

$$
\mathtt{translate}(\Xi, \mathtt{m}\, \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_k)\ \{\, \mathbb{c} \,\} : \mathbb{o},\ \mathtt{R}) \; = \;
\begin{cases}
\mathtt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = 0 & [\alpha_1 = \bot] \\
\mathtt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{1,1} & [\mathit{se}_1 \wedge \alpha_1 \neq \bot] \\
\quad \vdots & \\
\mathtt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_1,1} & [\mathit{se}_1 \wedge \alpha_1 \neq \bot] \\
\mathtt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{1,2} & [\mathit{se}_2 \wedge \alpha_1 \neq \bot] \\
\quad \vdots & \\
\mathtt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_n,n} & [\mathit{se}_n \wedge \alpha_1 \neq \bot] \\
\mathtt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = 0 & [\alpha_1 = \bot] \\
\mathtt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = \mathtt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \ldots, \alpha_n]) & [\alpha_1 = \partial] \\
\mathtt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_1,1} & [\mathit{se}_1 \wedge \alpha_1 = \top] \\
\quad \vdots & \\
\mathtt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_n,n} & [\mathit{se}_n \wedge \alpha_1 = \top]
\end{cases}
$$

Let $(\mathbb{C}_1\ \ldots\ \mathbb{C}_n,\ \mathbb{c})$ be a behavioural type program and let $\mathtt{translate}(\varnothing, \varnothing, \alpha, (\mathtt{true})\{\, 0 \,\}, \mathbb{c}) = \bigcup_{j=1}^{m} (\mathit{se}_j)\{\, e_{1,j}; \cdots; e_{h_j,j} \,\}$. Then

$$
\mathtt{translate}(\mathbb{C}_1\ \ldots\ \mathbb{C}_n,\ \mathbb{c}) \; = \;
\begin{cases}
\mathtt{translate}(\mathbb{C}_1) \cdots \mathtt{translate}(\mathbb{C}_n) & \\
\mathtt{main}() = 1 + e_{1,1} & [\mathit{se}_1] \\
\quad \vdots & \\
\mathtt{main}() = 1 + e_{h_1,1} & [\mathit{se}_1] \\
\mathtt{main}() = 1 + e_{1,2} & [\mathit{se}_2] \\
\quad \vdots & \\
\mathtt{main}() = 1 + e_{h_m,m} & [\mathit{se}_m]
\end{cases}
$$

As an example, we show the output of `translate` when applied to the behavioural type of `double_release` computed in Section 4. Since `double_release` has two arguments, we generate two sets of equations, as discussed above. In order to ease the reading, we omit the equivalence classes of arguments that label function names: the reader may grasp them from the number of arguments. For the same reason, we represent a partition $\{\{1\}, \{2\}, \{3\}\}$ corresponding to vm names α_1, α_2 and α_3 by $[\alpha_1, \alpha_2, \alpha_3]$ and $\{\{1\}, \{2, 3\}\}$ by $[\alpha_1, \alpha_2]$ (we write the canonical representatives). For simplicity we do not add the partition to the name of the method.

$$
\mathtt{translate}([\alpha_1, \alpha_2, \alpha_3], \mathit{double\_release}\ \alpha_1(\alpha_2, \alpha_3)\ \{\, \mathbb{c} \,\} : \_,\ \{\alpha_2, \alpha_3\})
$$

$$
= \begin{cases}
\mathit{double\_release}_{\text{peak}}(\alpha_1, \alpha_2, \alpha_3) = 0 & [\alpha_1 = \bot] \\
\mathit{double\_release}_{\text{peak}}(\alpha_1, \alpha_2, \alpha_3) = 0 & [\alpha_1 \neq \bot] \\
\mathit{double\_release}_{\text{peak}}(\alpha_1, \alpha_2, \alpha_3) = \mathtt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
\mathit{double\_release}_{\text{peak}}(\alpha_1, \alpha_2, \alpha_3) = \mathtt{CREL}(\alpha_2) + \mathtt{CREL}(\alpha_3) & [\alpha_1 \neq \bot] \\[6pt]
\mathit{double\_release}_{\text{net}}(\alpha_1, \alpha_2, \alpha_3) = 0 & [\alpha_1 = \bot] \\
\mathit{double\_release}_{\text{net}}(\alpha_1, \alpha_2, \alpha_3) = \mathit{double\_release}_{\text{peak}}(\alpha_1, \alpha_2, \alpha_3) & \\
 & [\alpha_1 = \partial] \\
\mathit{double\_release}_{\text{net}}(\alpha_1, \alpha_2, \alpha_3) = \mathtt{CREL}(\alpha_2) + \mathtt{CREL}(\alpha_3) & [\alpha_1 = \top]
\end{cases}
$$

$$\texttt{translate}([\alpha_1,\alpha_2], \mathit{double\_release}\ \alpha_1(\alpha_2)\ \{\,\mathbb{c}\,\} : \_,\ \{\alpha_2\}) \ =$$

$$\begin{cases}
\mathit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\
\mathit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 \neq \bot] \\
\mathit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
\mathit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 \neq \bot] \\
 & \\
\mathit{double\_release}_{\text{net}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\
\mathit{double\_release}_{\text{net}}(\alpha_1,\alpha_2) = \mathit{double\_release}_{\text{peak}}(\alpha_1,\alpha_2) & [\alpha_1 = \partial] \\
\mathit{double\_release}_{\text{net}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 = \top]
\end{cases}$$

To highlight a cost computation concerning `double_release`, consider the following two potential users

```
Int user1() {
 Vm x ; Vm y ; Fut<Int> f ;
 x = new Vm ; y = new Vm;
 f = this!double_release(x, y);
 f.get ; return 0 ; }
```

```
Int user2() {
 Vm x ; Fut<Int> f ;
 Vm x = new Vm ;
 f = this!double_release(x, x);
 f.get ; return 0 ; }
```

which have corresponding behavioural types

$$\begin{array}{l}
\texttt{user1}\ \alpha(\ )\{ \\
\quad \nu\beta\ ⨾\ \nu\gamma\ ⨾ \\
\quad \nu\, f : \texttt{double\_release}\, \alpha(\beta,\gamma)\ ⨾\ f^{\checkmark} \\
\}\ \texttt{-}\ ,\ \{\ \}
\end{array}$$

$$\begin{array}{l}
\texttt{user2}\ \alpha(\ )\{ \\
\quad \nu\beta\ ⨾ \\
\quad \nu\, f : \texttt{double\_release}\, \alpha(\beta,\beta)\ ⨾\ f^{\checkmark} \\
\}\ \texttt{-}\ ,\ \{\ \}
\end{array}$$

The translations of the foregoing types give the set of equations

$$\texttt{translate}([\alpha_1], \mathit{user1}\ \alpha_1()\ \{\,\mathbb{c}_{\mathit{user1}}\,\} : \_,\ \{\}) \ =$$

$$\begin{cases}
\mathit{user1}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & \\
\qquad\qquad + \mathit{double\_release}_{\text{peak}}(\alpha_1,\top,\top) & [\alpha_1 \neq \bot] \\
\mathit{user1}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & \\
\qquad\qquad + \mathit{double\_release}_{\text{net}}(\alpha_1,\top,\top) & [\alpha_1 \neq \bot] \\
 & \\
\mathit{user1}_{\text{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user1}_{\text{net}}(\alpha_1) = \mathit{user1}_{\text{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
\mathit{user1}_{\text{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & \\
\qquad\qquad + \mathit{double\_release}_{\text{net}}(\alpha_1,\top,\top) & [\alpha_1 = \top]
\end{cases}$$

$$\texttt{translate}([\alpha_1], \mathit{user2}\ \alpha_1()\ \{\,\mathbb{c}_{\mathit{user2}}\,\} : \_,\ \{\}) \ =$$

$$\begin{cases}
\mathit{user2}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\text{peak}}(\alpha_1,\top) & [\alpha_1 \neq \bot] \\
\mathit{user2}_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\text{net}}(\alpha_1,\top) & [\alpha_1 \neq \bot] \\
 & \\
\mathit{user2}_{\text{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
\mathit{user2}_{\text{net}}(\alpha_1) = \mathit{user2}_{\text{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
\mathit{user2}_{\text{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \mathit{double\_release}_{\text{net}}(\alpha_1,\top) & [\alpha_1 = \top]
\end{cases}$$

If we compute the cost of $\mathit{user1}_{\text{peak}}(\alpha)$ and $\mathit{user1}_{\text{net}}(\alpha)$ we obtain 2 and 0, respectively. That is, in this case, `double_release` being invoked with two different arguments has cost `-2`. On the contrary, the cost of $\mathit{user2}_{\text{peak}}(\alpha)$ and $\mathit{user2}_{\text{net}}(\alpha)$ is `1` and `0`, respectively. That is, in this case, `double_release` being invoked with two equal arguments has cost `-1`.

6. Outline of the proof of correctness

The proof of correctness of our technique is long even if almost standard (see [12] for a similar proof). In this section we overview it by highlighting the main difficulties.

The first part of the proof addresses the correctness of the type system in Section 4. As usual with type systems, the correctness is represented by a subject reduction theorem expressing that if a configuration *cn* of the operational semantics is well typed and *cn* $\to$ *cn*$'$ then *cn*$'$ is well-typed as well. It is worth to observe that we cannot hope to demonstrate a statement guaranteeing type-preservation because our types are "behavioural" and change during the evolution of the systems. However, it is critical for the correctness of the cost analysis that there exists a relation between the type of *cn*, let it be $\mathbb{c}$, and the type of *cn*$'$, let it be $\mathbb{c}'$.

Therefore, a subject reduction for the type system of Section 4 requires

1. the extension of the typing to configurations;
2. the definition of an evaluation relation $\rightsquigarrow$ between behavioural types.

Once 1 and 2 above have been defined, it is possible to demonstrate (let $\rightsquigarrow^*$ be the reflexive and transitive closure of $\rightsquigarrow$):

Theorem 6.1 (Subject Reduction)**.** *Let cn be a configuration of a* `vml` *program and let* $\mathbb{c}$ *be its behavioural type. If cn* $\to$ *cn*$'$ *then there is* $\mathbb{c}'$ *typing cn*$'$ *such that* $\mathbb{c} \rightsquigarrow^* \mathbb{c}'$.

The proof of this theorem is by case on the reduction rule applied and it is usually not complex because the relation $\rightsquigarrow$ mimics the `vml` transitions in Section 2.

The second part of the proofs relies on the definition of the notion of *direct cost of a behavioural type* (of a configuration), which is the number of virtual machines occurring in the type. The basic remark here is that the number of alive virtual machines in a configuration is identical to the direct cost of the corresponding a behavioural type. This requires

3. the extension of the function `translate` to compute the cost equations for behavioural types of configurations. These equations allow us to compute the *peak cost of a behavioural type* (of a configuration).

The proofs of the following two properties are preliminary to the correctness of our technique:

Lemma 6.2 (Basic Cost Inclusion)**.** *The direct cost of a behavioural type of a configuration is less or equal to its peak cost.*

Lemma 6.3 (Reduction Cost Inclusion). *If* $\mathbb{c} \rightsquigarrow \mathbb{c}'$ *then the peak cost of* $\mathbb{c}'$ *is less or equal to the peak cost of* $\mathbb{c}$.

It is important to observe that the proofs of Lemmas 6.2 and 6.3 are given using the (theoretical) solution of cost equations in [11]. This lets us to circumvent possible errors in implementations of the theory, such as `CoFloCo` [11] or `PUBS` [1]. Given the basic cost and reduction cost inclusions, we can demonstrate the correctness theorem for our technique.

Theorem 6.4 (Correctness)**.** *Let* $\overline{M}\ \ \{\overline{F\,z}\ \ ;\ s'\}$ *be a well-typed program and let* $\overline{\mathbb{C}}$, $\mathbb{c}$ *be its behavioural type. Let also* n *be a solution of the function* $\texttt{translate}(\overline{\mathbb{C}},\ \mathbb{c})$. *Then* n *is an upper bound of the number of virtual machines used during the execution of cn.*

The proof outline is as follows. Since the cost of the initial configuration *cn* is the direct cost of $\mathbb{c}$ then, by Lemma 6.2, this value is less or equal to the peak cost of $\mathbb{c}$. Let n be a solution of this cost. The argument proceeds by induction on the number of reduction steps:

- for the base case, when the program doesn't reduce, it turns out that $n \geq 1$;
- for the inductive case, let *cn* $\to$ *cn*$'$. By applying Theorem 6.1 and Lemma 6.3, one derives that n is bigger than the peak cost of the behavioural type of *cn*$'$. Thus, by Lemma 6.2, we have that n is larger than the number of alive virtual machines in *cn*$'$.

7. Integration with a cost analysis tool

We briefly discuss technical details about the translation of the recurrence relations in Section 5 into the CoFloCo analyser [11] and we examine the outputs obtained by running examples of this paper.

In order to comply with usual input formats of tools, we need to define encodings for vm values and for the functions CNEW and CREL. We therefore define

- $\top$ is modelled by 1, ∂ is modelled by 2, $\bot$ is modelled by 3, and α by α. As regards $\alpha\downarrow$, it is modelled by the conditional value $[\alpha = 3]3 + [1 \leq \alpha \leq 2]2$;
- the auxiliary functions CNEW and CREL are translated in recurrence relations as follows:

```
eq(CNEW(A), 0, [], [A = 3]).
eq(CNEW(A), 1, [], [A < 3]).

eq(CREL(A), -1, [], [A = 1]).
eq(CREL(A), 0, [], [A > 1]).
```

We begin our experiments with the translation of double_release when used by user1. In this case, the arguments of double_release are all different, therefore we use $double\_release_{\text{peak}}(\alpha_1, \alpha_2, \alpha_3)$ and $double\_release_{\text{net}}(\alpha_1, \alpha_2, \alpha_3)$. We write these functions in CoFloCo as doubleRelease123_peak(A,B,C) and doubleRelease123_net(A,B,C). The input code for the cost analyser is omitted. The following table report the output of CoFloCo

Entry Point	Cost
entry(user1_net(1):[]).	0
entry(user1_peak(1):[]).	2

which is exactly what we anticipated in Section 5.

When double_release is invoked used by user2 then its arguments are equal. That is $double\_release_{\text{peak}}(\alpha_1, \alpha_2)$ and $double\_release_{\text{net}}(\alpha_1, \alpha_2)$ are used, which are written in CoFloCo as doubleRelease12_peak(A,B) and doubleRelease12_net(A,B). The table below shows the output of the cost analyzer for the given equations, where, again, we consider only the case when the first argument is alive, that is, it is equal to 1.

Entry Point	Cost
entry(user2_net(1):[]).	0
entry(user2_peak(1):[]).	1

As before, the cost is exactly what computed in Section 5.

8. Related Work

After the pioneering work by Wegbreit in 1975 [21] that discussed a method for deriving upper-bounds costs of functional programs, a number of cost analysis techniques have been developed. Those ones that are closely related to this contribution are based either on cost equations (solvers) or on amortized analysis.

The techniques based on cost equations address cost analysis in three steps by: (*i*) extracting relevant information out of the original programs by abstracting data structures to their size and assigning a cost to every program expression, (*ii*) converting the abstract program into cost equations, and (*iii*) solving the cost equations with an automatic tool. Recent advances have been done for improving the accuracy of upper-bounds for cost equations [1, 4, 9, 11, 13] and we refer to [11] for a comparison of these tools. The main advantage of these techniques is that cost equations may carry Presburger arithmetic conditions thus supporting a precise cost analysis of conditional statements. The main drawback is that they extract control flow graphs from programs to perform their analysis, using abstract interpretations and control flow refinement techniques [4, 5]. It turns out that the above techniques do not provide the alias analysis and the name identity management that we have done in Section 4, which are essential for function or procedure abstraction, thus jeopardising compositional reasoning when large programs are considered.

The techniques based on amortized analysis [20] associate so-called potentials to program expressions by means of type systems (these potential determine the resources needed for each expression to be evaluated). The connection between the original program and the cost equations can be indeed demonstrated by a standard subject-reduction theorem [10, 15–18]. While the techniques based on types are intrinsically compositional and, more importantly, type derivations can be seen as certificates of abstract descriptions of functions, type based methods do not model the interaction of integer arithmetic with resource usage, thus being less accurate in some cases. An emblematic example is the following function:

```
Int foo(Int n, Int m, Vm x) {
  Fut<Int> f ;
  if (n==0) return 0;
  else if (n>m) { Vm v = new Vm ;
                  f = x!foo(n-1,m,v) ; f.get ; return 0 ;
  } else { Vm v = new Vm ; Vm w = new Vm ;
           f = v!foo(n-1,m,w) ; f.get ; return 0 ; }
}

{
  Vm x = new Vm ; Fut<Int> f = this!foo(2*n, n, x) ;
  f.get ;
}
```

which recursively invokes itself 2*n times, and half the times executes the second branch – the case (n>m) – with cost 1 and half the times executes the else branch (with cost 2). The techniques based on amortized analysis give a cost for the function foo that is 2*(2*n)=4*n – without recognizing that the most costly branch is executed only half of the times – because they always assign the same cost to every branch. On the contrary, a more accurate analysis should derive that the actual cost of foo is 2*n + n = 3*n. In fact, this is the case of solvers based cost equations, such as [11].

The technique proposed in this paper combines the advantages of the two approaches discussed above. It is modular, like the techniques based on cost equations, our one also consists of three steps, and it extracts the relevant information of programs by means of a *behavioural* type system, like the technique based on amortized analysis. Therefore, our technique is compositional and can be proved sound by means of a standard subject-reduction theorem. At the same time it is accurate in modelling the interaction of integer arithmetic with resource usage.

A common feature of cost analysis techniques in the literature is that they analyze *cumulative resources*. That is, resources that *do not decrease* during the execution of the programs, such as execution time, number of operations, memory (without an explicit free operation). As already discussed, this assumption eases the analysis because it permits to compute over-approximated cost. On the contrary, the presence of an *explicit or implicit* release operation entangles the analysis. In [2], a memory cost analysis is proposed for languages with garbage collection. It is worth to say that the setting of [2] is not difficult because, by definition of garbage collection, released memory is always inactive. The impact of the release operation in the cost analysis is thoroughly discussed in [7] by means of the notions of *peak cost* and *net cost* that we have also used in Section 5. It is worth to notice that, for cumulative analysis, this two notions coincide while, in non-cumulative analysis (in presence of a release operation), they are different and the *net cost* is key for computing tight upper bounds.

Recently [6] has analysed the cost of a language with explicit releases. We observe that the release operation studied in [6] is used in a very restrictive way: only locally created resources can be released. This constraint guarantees that costs of functions are always not negative, thus permitting the (re)use of non-negative cost models of cumulative analysis.

We conclude by discussing cost analysis techniques for concurrent systems, which are indeed very few [3, 5, 14]. In order to reduce the imprecision of the analysis caused by the nondeterminism, [3, 5] use a clever technique for isolating sequential code from parallel code, called *may-happen-in-parallel* [8]. We notice that no one of these contributions consider a concurrent language with a powerful `release` operation that allows one remove the resources taken in input. In fact, without this operation, one can model the cost by simply aggregating the sets of operations that can occur in parallel, as in [5], and all the theoretical development is much easier.

9. Conclusion

This paper presents the first (to the best of our knowledge) static analysis technique that computes upper bound of virtual machines usages in concurrent programs that may create and, more importantly, may release such machines. Our analysis consists of a type system that extracts relevant information about resource usages in programs, called behavioural types; an automatic translation that transforms these types into cost expressions; the application of solvers, like `CoFloCo` [11], on these expressions that compute upper bounds of the usage of virtual machines in the original program. A relevant property of our technique is its modularity. For the sake of simplicity, we have applied the technique to a small language. However, by either extending or changing the type system, the analysis can be applied to many other languages with primitives for creating and releasing resources. In addition, by changing the translation algorithm, it is possible to target other solvers that may compute better upper bounds.

For the future, we consider at least two lines of work. First, we intend to alleviate the restrictions introduced in Section 3 on the programs we can analyse. This may be pursued by retaining more expressive notations for the effect of a method, *i.e.* by considering `R` as a set of sets instead of a simple set. Such a notation is more suited for modelling nondeterministic behaviours and it might be made even more expressive by tagging all the different effects in `R` with a condition specifying when such effect is yielded. Clearly, the management of these domains becomes more complex and the trade-off between simplicity and expressiveness must be carefully evaluated. Second, we intend to implement our analysis targeting a programming language with a formal model as `ABS` [19], of which `vml` is a very basic sub-calculus. The current prototype translates a behavioural type program into cost equations and we expect to define an inference system that returns behavioural types in the same style of [12].

Acknowledgments

We thank Elena Giachino for the valuable discussions and for the help in the assessment of our technique with respect to those based on amortized analysis. We also thank Antonio Flores-Montoya and Samir Genaim for the discussions about the cost analysers `CoFloCo` and `PUBS`.

References

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

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. *SIGPLAN Not.*, 45(8):121–130, 2010.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost analysis of concurrent OO programs. In *Proceedings of APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2011.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[5] E. Albert, J. Correas, and G. Romn-Dez. Peak cost analysis of distributed systems. In *Proceedings of SAS 2014*, volume 8723 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014.

[6] E. Albert, J. Correas, and G. Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of TACAS 2015*, Lecture Notes in Computer Science. Springer, 2015. To appear.

[7] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *Proceedings of SAS 2012*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2012.

[8] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *Proceedings of LCPC 2005*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2006.

[9] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proceedings of TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[10] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *Proceedings of SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005.

[11] A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of 12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[12] E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Software and Systems Modeling*, 2015. ISSN 1619-1366. . URL `http://dx.doi.org/10.1007/s10270-014-0444-y`. To appear.

[13] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139. ACM, 2009.

[14] J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs, 2015. URL `http://cs.yale.edu/homes/hoffmann/papers/parallelcost2014.pdfm`. [Online; accessed 11-February-2015].

[15] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012. . URL `http://doi.acm.org/10.1145/2362389.2362393`.

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

[17] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *Proceedings of ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[18] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proceedings of CSL 2009*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009.

[19] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[20] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[21] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

Appendix C

Time complexity of concurrent programs

Time complexity of concurrent programs[⋆]
– a technique based on behavioural types –

Elena Giachino[1], Einar Broch Johnsen[2], Cosimo Laneve[1], and Ka I Pun[2]

[1] Dept. of Computer Science and Engineering, University of Bologna – INRIA FOCUS
[2] Dept. of Informatics, University of Oslo

Abstract. We study the problem of automatically computing the time complexity of concurrent object-oriented programs. To determine this complexity we use intermediate abstract descriptions that record relevant information for the time analysis (cost of statements, creations of objects, and concurrent operations), called *behavioural types.* Then, we define a translation function that takes behavioural types and makes the parallelism explicit into so-called *cost equations*, which are fed to an automatic off-the-shelf solver for obtaining the time complexity.

1 Introduction

Computing the cost of a sequential algorithm has always been a primary question for every programmer, who learns the basic techniques in the first years of their computer science or engineering curriculum. This cost is defined in terms of the input values to the algorithm and over-approximates the number of the executed instructions. In turn, given an appropriate abstraction of the CPU speed of a runtime system, one can obtain the expected computation time of the algorithm.

The computational cost of algorithms is particularly relevant in mainstream architectures, such as the cloud. In that context, a service is a concurrent program that must comply with a so-called *service-level agreement* (SLA) regulating the cost in time and assigning penalties for its infringement [3]. The service provider needs to make sure that the service is able to meet the SLA, for example in terms of the end-user response time, by deciding on a resource management policy and determining the appropriate number of virtual machine instances (or containers) and their parameter settings (e.g., their CPU speeds). To help service providers make correct decisions about the resource management before actually deploying the service, we need static analysis methods for resource-aware services [6]. In previous work by the authors, cloud deployments expressed in the formal modeling language ABS [8] have used a combination of cost analysis and simulations to analyse resource management [1], and a Hoare-style proof system to reason about end-user deadlines has been developed for sequential executions [7]. In contrast, we are here interested in statically estimating the

[⋆] Supported by the EU projects FP7-610582 Envisage: Engineering Virtualized Services (`http://www.envisage-project.eu`).

computation time of concurrent services deployed on the cloud with a given dynamic resource management policy.

Technically, this paper proposes a behavioural type system expressing the resource costs associated with computations and study how these types can be used to soundly calculate the time complexity of parallel programs deployed on the cloud. To succinctly formulate this problem, our work is developed for `tml`, a small formally defined concurrent object-oriented language which uses asynchronous communications to trigger parallel activities. The language defines virtual machine instances in terms of dynamically created concurrent object groups with bounds on the number of cycles they can perform per time interval. As we are interested in the concurrent aspects of these computations, we abstract from sequential analysis in terms of a statement $\texttt{job}(e)$, which defines the number of processing cycles required by the instruction – this is similar to the `sleep(n)` operation in `Java`.

The analysis of behavioural types is defined by translating them in a code that is adequate for an off-the-shelf solver – the `CoFloCo` solver [4]. As a consequence, we are able to determine the computational cost of algorithms in a parametric way with respect to their inputs.

Paper overview. The language is defined in Section 2 and we discuss restrictions that ease the development of our technique in Section 3. Section 4 presents the behavioural type system and Section 5 explains the analysis of computation time based on these behavioural types. In Section 6 we outline our correctness proof of the type system with respect to the cost equations. In Section 7 we discuss the relevant related work and in Section 8 we deliver concluding remarks.

2 The language `tml`

The syntax and the semantics of `tml` are defined in the following two subsections; the third subsection discusses a few examples.

Syntax. A `tml` program is a sequence of method definitions $T\ \texttt{m}(\overline{T\ x})\{\ \overline{F\ y\ ;}\ s\ \}$, ranged over by M, plus a main body $\{\ \overline{F\,z\ ;}\ s'\ \}\ \texttt{with}\ k$. In `tml` we distinguish between *simple types* T which are either integers `Int` or classes `Class` (there is just one class in `tml`), and *types* F, which also include *future types* `Fut<`T`>`. These future types let asynchronous method invocations be typed (see below). The notation $\overline{T\ x}$ denotes any finite sequence of *variable declarations* $T\ x$. The elements of the sequence are separated by commas. When we write $\overline{T\ x\ ;}$ we mean a sequence $T_1\ x_1\ ;\ \cdots\ ;\ T_n\ x_n\ ;$ when the sequence is not empty; we mean the possibly empty sequence otherwise.

The syntax of statements s, expressions with side-effects z and expressions e of `tml` is defined by the following grammar:

$$\begin{array}{l}
s ::= x = z \ \mid\ \texttt{if}\ e\ \{\,s\,\}\ \texttt{else}\ \{\,s\,\}\ \mid\ \texttt{job}(e)\ \mid\ \texttt{return}\ e\ \mid\ s\ ;\ s\\
z ::= e\ \mid\ e\texttt{!m}(\overline{e})\ \mid\ e.\texttt{m}(\overline{x})\ \mid\ e.\texttt{get}\ \mid\ \texttt{new Class with}\ e\ \mid\ \texttt{new local Class}\\
e ::= \texttt{this}\ \mid\ se\ \mid\ nse
\end{array}$$

A statement s may be either one of the standard operations of an imperative language or the job statement `job`(e) that delays the continuation by e cycles of the machine executing it.

An expression z may change the state of the system. In particular, it may be an *asynchronous* method invocation of the form e`!m`$(\overline{e})$, which does not suspend the caller's execution. When the value computed by the invocation is needed, the caller performs a *non-blocking* `get` operation: if the value needed by a process is not available, then an awaiting process is scheduled and executed, i.e., *await-get*. Expressions z also include standard synchronous invocations e.`m`$(\overline{e})$ and `new local Class`, which creates a new object. The intended meaning is to create the object in the same machine – called *cog* or *concurrent object group* – of the caller, thus sharing the processor of the caller: operations in the same virtual machine interleave their evaluation (even if in the following operational semantics the parallelism is not explicit). Alternatively, one can create an object on a different cog with `new Class with` e thus letting methods execute in parallel. In this case, e represents the capacity of the new cog, that is, the number of cycles the cog can perform per time interval. We assume the presence of a special identifier `this.capacity` that returns the capacity of the corresponding cog.

A *pure* expression e can be the reserved identifier `this` or an integer expression. Since the analysis in Section 5 cannot deal with generic integer expressions, we parse expressions in a careful way. In particular we split them into *size expressions se*, which are expressions in Presburger arithmetics (this is a decidable fragment of Peano arithmetics that only contains addition), and *non-size expressions nse*, which are the other type of expressions. The syntax of size and non-size expressions is the following:

$$\begin{array}{rl}
\mathit{nse} ::= & k \mid x \mid \mathit{nse} \le \mathit{nse} \mid \mathit{nse}\ \texttt{and}\ \mathit{nse} \mid \mathit{nse}\ \texttt{or}\ \mathit{nse} \\
& \mid \mathit{nse} + \mathit{nse} \mid \mathit{nse} - \mathit{nse} \mid \mathit{nse} \times \mathit{nse} \mid \mathit{nse}/\mathit{nse} \\
\mathit{se} ::= & \mathit{ve} \mid \mathit{ve} \le \mathit{ve} \mid \mathit{se}\ \texttt{and}\ \mathit{se} \mid \mathit{se}\ \texttt{or}\ \mathit{se} \\
\mathit{ve} ::= & k \mid x \mid \mathit{ve} + \mathit{ve} \mid k \times \mathit{ve} \\
k ::= & \textit{rational constants}
\end{array}$$

In the paper, we assume that sequences of declarations $\overline{T\ x}$ and method declarations $\overline{M}$ do not contain duplicate names. We also assume that `return` statements have no continuation.

Semantics. The semantics of `tml` is defined by a transition system whose states are *configurations cn* that are defined by the following syntax.

$$\begin{array}{rlrl}
\mathit{cn} ::= & \varepsilon \mid \mathit{fut}(f, \mathit{val}) \mid \mathit{ob}(o, c, p, q) \mid \mathit{invoc}(o, f, \texttt{m}, \overline{v}) & \mathit{act} ::= & o \mid \varepsilon \\
& \mid \mathit{cog}(c, \mathit{act}, k) \mid \mathit{cn}\ \mathit{cn} & \mathit{val} ::= & v \mid \bot \\
p ::= & \{\, l \mid s \,\} \mid \texttt{idle} & l ::= & [\cdots, x \mapsto v, \cdots] \\
q ::= & \varnothing \mid \{\, l \mid s \,\} \mid q\ q & v ::= & o \mid f \mid k
\end{array}$$

A *configuration cn* is a set of concurrent object groups (cogs), objects, invocation messages and futures, and the empty configuration is written as ε. The associative and commutative union operator on configurations is denoted by whitespace. A *cog* is given as a term $\mathit{cog}(c, \mathit{act}, k)$ where c and k are respectively the identifier and the capacity of the cog, and *act* specifies the currently active

(Cond-True)
$$\frac{\texttt{true} = \llbracket e \rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l \mid s_1\ \texttt{;}\ s \,\}, q) \end{array}}$$

(Cond-False)
$$\frac{\texttt{false} = \llbracket e \rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l \mid s_2\ \texttt{;}\ s \,\}, q) \end{array}}$$

(New)
$$\frac{c' = \text{fresh}(\,) \quad o' = \text{fresh}(\,) \quad k = \llbracket e \rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid x = \texttt{new Class with}\ e\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l \mid x = o'\ \texttt{;}\ s \,\}, q) \\ ob(o', c', \texttt{idle}, \varnothing) \quad cog(c', o', k) \end{array}}$$

(New-Local)
$$\frac{o' = \text{fresh}(\,)}{\begin{array}{c} ob(o,c,\{\, l \mid x = \texttt{new local Class}\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l \mid x = o'\ \texttt{;}\ s \,\}, q) \\ ob(o', c, \texttt{idle}, \varnothing) \end{array}}$$

(Get-True)
$$\frac{f = \llbracket e \rrbracket_l \quad v \neq \bot}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{get}\ \texttt{;}\ s \,\}, q) \quad fut(f, v) \\ \to ob(o,c,\{\, l \mid x = v\ \texttt{;}\ s \,\}, q) \quad fut(f, v) \end{array}}$$

(Get-False)
$$\frac{f = \llbracket e \rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{get}\ \texttt{;}\ s \,\}, q) \quad fut(f, \bot) \\ \to ob(o,c,\texttt{idle}, q \cup \{\, l \mid x = e.\texttt{get}\ \texttt{;}\ s \,\}) \quad fut(f, \bot) \end{array}}$$

(Self-Sync-Call)
$$\frac{\begin{array}{c} o = \llbracket e \rrbracket_l \quad \overline{v} = \llbracket \overline{e} \rrbracket_l \quad f' = l(\texttt{destiny}) \\ f = \text{fresh}(\,) \quad \{\, l' \mid s' \,\} = \text{bind}(o, f, \texttt{m}, \overline{v}) \end{array}}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{m}(\overline{e})\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l' \mid s'\ \texttt{;}\ \texttt{cont}(f') \,\}, q \cup \{\, l \mid x = f.\texttt{get}\ \texttt{;}\ s \,\}) \\ fut(f, \bot) \end{array}}$$

(Self-Sync-Return-Sched)
$$\frac{f = l'(\text{destiny})}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{cont}(f) \,\}, q \cup \{\, l' \mid s \,\}) \\ \to ob(o,c,\{\, l' \mid s \,\}, q) \end{array}}$$

(Cog-Sync-Call)
$$\frac{\begin{array}{c} o' = \llbracket e \rrbracket_l \quad \overline{v} = \llbracket \overline{e} \rrbracket_l \quad f' = l(\texttt{destiny}) \\ f = \text{fresh}(\,) \quad \{\, l' \mid s' \,\} = \text{bind}(o', f, \texttt{m}, \overline{v}) \end{array}}{\begin{array}{c} ob(o,c,\{\, l \mid x = e.\texttt{m}(\overline{e})\ \texttt{;}\ s \,\}, q) \\ ob(o', c, \texttt{idle}, q') \quad cog(c, o, k) \\ \to ob(o,c,\texttt{idle}, q \cup \{\, l \mid x = f.\texttt{get}\ \texttt{;}\ s \,\}) \quad fut(f, \bot) \\ ob(o', c, \{\, l' \mid s'\ \texttt{;}\ \texttt{cont}(f') \,\}, q') \quad cog(c, o', k) \end{array}}$$

(Cog-Sync-Return-Sched)
$$\frac{f = l'(\text{destiny})}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{cont}(f) \,\}, q) \quad cog(c, o, k) \\ ob(o', c, \texttt{idle}, q' \cup \{\, l' \mid s' \,\}) \\ \to ob(o,c,\texttt{idle}, q) \quad cog(c, o', k) \\ ob(o', c, \{\, l' \mid s' \,\}, q') \end{array}}$$

(Async-Call)
$$\frac{o' = \llbracket e \rrbracket_l \quad \overline{v} = \llbracket \overline{e} \rrbracket_l \quad f = \text{fresh}(\,)}{\begin{array}{c} ob(o,c,\{\, l \mid x = e!\texttt{m}(\overline{e})\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l \mid x = f\ \texttt{;}\ s \,\}, q) \quad invoc(o', f, \texttt{m}, \overline{v}) \quad fut(f, \bot) \end{array}}$$

(Bind-Mtd)
$$\frac{\{\, l \mid s \,\} = \text{bind}(o, f, \texttt{m}, \overline{v})}{\begin{array}{c} ob(o,c,p,q) \quad invoc(o, f, \texttt{m}, \overline{v}) \\ \to ob(o,c,p,q \cup \{\, l \mid s \,\}) \end{array}}$$

(Context)
$$\frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}$$

(Release-Cog)
$$\begin{array}{c} ob(o,c,\texttt{idle},q) \quad cog(c,o,k) \\ \to ob(o,c,\texttt{idle},q) \quad cog(c,\varepsilon,k) \end{array}$$

(Activate)
$$\begin{array}{c} ob(o,c,\texttt{idle},q \cup \{\, l \mid s \,\}) \quad cog(c,\varepsilon,k) \\ \to ob(o,c,\{\, l \mid s \,\},q) \quad cog(c,o,k) \end{array}$$

(Return)
$$\frac{v = \llbracket e \rrbracket_l \quad f = l(\text{destiny})}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{return}\ e \,\}, q) \quad fut(f, \bot) \\ \to ob(o,c,\texttt{idle},q) \quad fut(f, v) \end{array}}$$

(Job-0)
$$\frac{\llbracket e \rrbracket_l = 0}{\begin{array}{c} ob(o,c,\{\, l \mid \texttt{job}(e)\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l \mid s \,\}, q) \end{array}}$$

(Assign-Local)
$$\frac{x \in \text{dom}(l) \quad v = \llbracket e \rrbracket_l}{\begin{array}{c} ob(o,c,\{\, l \mid x = e\ \texttt{;}\ s \,\}, q) \\ \to ob(o,c,\{\, l\,[x \mapsto v] \mid s \,\}, q) \end{array}}$$

Fig. 1. The transition relation of `tml` – part 1.

object in the cog. An object is written as $ob(o, c, p, q)$, where o is the identifier of the object, c the identifier of the cog the object belongs to, p an *active process*, and q a pool of *suspended processes*. A *process* is written as $\{\, l \mid s \,\}$, where l denotes local variable bindings and s a list of statements. An *invocation message* is a term $invoc(o, f, \texttt{m}, \overline{v})$ consisting of the callee o, the future f to which the result of the call is returned, the method name m, and the set of actual parameter values for the call. A *future* $fut(f, val)$ contains an identifier f and a reply value val, where $\bot$ indicates the reply value of the future has not been received.

The following auxiliary function is used in the semantic rules for invocations. Let $T'\ \texttt{m}(\overline{T\ x})\{\, \overline{F\ x'}; s \,\}$ be a method declaration. Then

$$\text{bind}(o, f, \texttt{m}, \overline{v}) = \{\, [\text{destiny} \mapsto f, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \bot] \mid s\{{}^{o}/_{\texttt{this}}\} \,\}$$

$$\text{(Tick)}\quad \frac{\mathit{strongstable}_t(cn)}{cn \rightarrow \Phi(cn, t)}$$

where

$$\Phi(cn,t) = \begin{cases} ob(o,c,\{l' \mid \texttt{job}(k')\ ;\ s\},q)\ \Phi(cn',t) & \text{if } cn = ob(o,c,\{l \mid \texttt{job}(e)\ ;\ s\},q)\ cn' \\ & \text{and } cog(c,o,k) \in cn' \\ & \text{and } k' = [\![e]\!]_l - k * t \\ ob(o,c,\texttt{idle},q)\ \Phi(cn',t) & \text{if } cn = ob(o,c,\texttt{idle},q)\ cn' \\ cn & \text{otherwise.} \end{cases}$$

Fig. 2. The transition relation of `tml` – part 2: the strongly stable case

The *transition rules* of `tml` are given in Figures 1 and 2. We discuss the most relevant ones: object creation, method invocation, and the `job`(e) operator. The creation of objects is handled by rules New and New-Local: the former creates a new object inside a new cog with a given capacity e, the latter creates an object in the local cog. Method invocations can be either synchronous or asynchronous. Rules Self-Sync-Call and Cog-Sync-Call specify synchronous invocations on objects belonging to the same cog of the caller. Asynchronous invocations can be performed on every object.

In our model, the unique operation that consumes time is `job`(e). We notice that the reduction rules of Figure 1 are not defined for the `job`(e) statement, except the trivial case when the value of e is 0. This means that time does not advance while non-job statements are evaluated. When the configuration cn reaches a *stable* state, *i.e.,* no other transition is possible apart from those evaluating the `job`(e) statements, then the time is advanced by the minimum value that is necessary to let at least *one* process start. In order to formalize this semantics, we define the notion of stability and the *update operation* of a configuration cn (with respect to a time value t). Let $[\![e]\!]_l$ return the value of e when variables are bound to values stored in l.

Definition 1. *Let $t > 0$. A configuration cn is t-*stable*, written $\mathit{stable}_t(cn)$, if any object in cn is in one of the following forms:*

1. *$ob(o, c, \{\, l \mid \texttt{job}(e); s\,\}, q)$ with $cog(c, o, k) \in cn$ and $[\![e]\!]_l/k \geq t$,*
2. *$ob(o, c, \texttt{idle}, q)$ and*
   i. *either $q = \varnothing$,*
   ii. *or, for every $p \in q$, $p = \{\, l \mid x = e.\texttt{get}; s\,\}$ with $[\![e]\!]_l = f$ and $fut(f, \bot)$,*
   iii. *or, $cog(c, o', k) \in cn$ where $o \neq o'$, and o' satisfies Definition 1.1.*

A configuration cn is strongly t-stable*, written $\mathit{strongstable}_t(cn)$, if it is t-stable and there is an object $ob(o, c, \{\, l \mid \texttt{job}(e); s\,\}, q)$ with $cog(c, o, k) \in cn$ and $[\![e]\!]_l/k \ = \ t$.*

Notice that t-stable (and, consequently, strongly t-stable) configurations cannot progress anymore because every object is stuck either on a job or on unresolved get statements. The update of cn with respect to a time value t, noted $\Phi(cn, t)$ is defined in Figure 2. Given these two notions, rule TICK defines the time progress. The initial configuration of a program with main method $\{\, \overline{F\ x};\ s \,\}$ `with` k is

$$\begin{array}{l} ob(start, \mathrm{start}, \{\, [\mathrm{destiny} \mapsto f_{start}, \overline{x} \mapsto \bot] \mid s \,\}, \varnothing) \\ cog(\mathrm{start}, start, k) \end{array}$$

where start and *start* are special cog and object names, respectively, and f_{start} is a fresh future name. As usual, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

Examples. To begin with, we discuss the Fibonacci method. It is well known that the computational cost of its sequential recursive implementation is exponential. However, this is not the case for the parallel implementation. Consider

```
Int fib(Int n) {
       if (n<=1) { return 1; }
       else {  Fut<Int> f; Class z; Int m1; Int m2;
               job(1);
               z = new Class with this.capacity ;
               f = this!fib(n-1); g = z!fib(n-2);
               m1 = f.get; m2 = g.get;
               return  m1 + m2; } }
```

Here, the recursive invocation `fib(n-1)` is performed on the `this` object while the invocation `fib(n-2)` is performed on a new cog with the same capacity (i.e., the object referenced by `z` is created in a new cog set up with `this.capacity`), which means that it can be performed in parallel with the former one. It turns out that the cost of the following invocation is `n`.

```
Class z; Int m; Int x; x = 1;
z = new Class with x;
m = z.fib(n);
```

Observe that, by changing the line `x = 1;` into `x = 2;` we obtain a cost of `n/2`.

Our semantics does not exclude paradoxical behaviours of programs that perform infinite actions without consuming time (preventing rule TICK to apply), such as this one

```
Int foo() { Int m; m = this.foo(); return m; }
```

This kind of behaviours are well-known in the literature, (*cf.* Zeno behaviours) and they may be easily excluded from our analysis by constraining recursive invocations to be prefixed by a `job`(e)-statement, with a positive e. It is worth to observe that this condition is not sufficient to eliminate paradoxical behaviours. For instance the method below does not terminate and, when invoked with `this.fake(2)`, where `this` is in a cog of capacity 2, has cost 1.

```
Int fake(Int n) {
      Int m; Class x;
      x = new Class with 2*n; job(1); m = x.fake(2*n); return m; }
```

Imagine a parallel invocation of the method `Int one() { job(1); }` on an object residing in a cog of capacity 1. At each stability point the `job`(1) of the latter method will compete with the `job`(1) of the former one, which will win every time, since having a greater (and growing) capacity it will require always less time. So at the first stability point we get `job`$(1-1/2)$ (for the method `one`), then `job`$(1-1/2-1/4)$ and so on, thus this sum will never reach 0.

In the examples above, the statement `job`(e) is a cost annotation that specifies how many processing cycles are needed by the subsequent statement in the code. We notice that this operation can also be used to program a timer which suspends the current execution for e units of time. For instance, let

```
Int wait(Int n) { job(n); return 0; }
```

Then, invoking `wait` on an object with capacity 1

```
Class timer; Fut<Class> f; Class x;
timer = new Class with 1;
f = timer!wait(5); x = f.get;
```

one gets the suspension of the current thread for 5 units of time.

3 Issues in computing the cost of `tml` programs

The computation time analysis of `tml` programs is demanding. To highlight the difficulties, we discuss a number of methods.

```
Int wrapper(Class x) {
      Fut<Int> f; Int z;
      job(1) ; f = x!server(); z = f.get;
      return z; }
```

Method `wrapper` performs an invocation on its argument `x`. In order to determine the cost of `wrapper`, we notice that, if `x` is in the same cog of the carrier, then its cost is (assume that the capacity of the carrier is `1`): $1+\mathit{cost}($`server`$)$ because the two invocations are sequentialized. However, if the cogs of `x` and of the carrier are different, then we are not able to compute the cost because we have no clue about the state of the cog of `x`.

Next consider the following definition of `wrapper`

```
Int wrapper_with_log(Class x) {
      Fut<Int> f; Fut<Int> g; Int z;
      job(1) ; f = x!server(); g = x!print_log(); z = f.get;
      return z; }
```

In this case the wrapper also asks the server to print its log and this invocation is not synchronized. We notice that the cost of `wrapper_with_log` is not anymore $1+\mathit{cost}($`server`$)$ (assuming that `x` is in the same cog of the carrier) because `print_log` might be executed *before* `server`. Therefore the cost of `wrapper_with_log` is $1+\mathit{cost}($`server`$)+\mathit{cost}($`print_log`$)$.

Finally, consider the following wrapper that also logs the information received from the server on a new cog without synchronising with it:

```
Int wrapper_with_external_log(Class x) {
      Fut<Int> f; Fut<Int> g; Int z; Class y;
      job(1) ; f = x!server(); g = x!print_log(); z = f.get;
      y = new Class with 1;
      f = y!external_log(z);
      return z; }
```

What is the cost of `wrapper_with_external_log`? Well, the answer here is debatable: one might discard the cost of `y!external_log(z)` because it is useless for the value returned by `wrapper_with_external_log`, or one might count it because one wants to count every computation that has been triggered by a method in its cost. In this paper we adhere to the second alternative; however, we think that a better solution should be to return different cost for a method: a *strict cost*, which spots the cost that is necessary for computing the returned value, and an *overall cost*, which is the one computed in this paper.

Anyway, by the foregoing discussion, as an initial step towards the time analysis of `tml` programs, we simplify our analysis by imposing the following constraint:

– *it is possible to invoke methods on objects either in the same cog of the caller or on newly created cogs.*

The above constraint means that, if the callee of an invocation is one of the arguments of a method then it must be in the same cog of the caller. It also means that, if an invocation is performed on a returned object then this object must be in the same cog of the carrier. We will enforce these constraints in the typing system of the following section – see rule T-Invoke.

4 A behavioural type system for `tml`

In order to analyse the computation time of `tml` programs we use abstract descriptions, called *behavioural types*, which are intermediate codes highlighting the features of `tml` programs that are relevant for the analysis in Section 5. These abstract descriptions support compositional reasoning and are associated to programs by means of a type system. The syntax of behavioural types is defined as follows:

$$\begin{array}{rll}
\mathbb{t} ::= & - \mid se \mid c[se] & \text{basic value} \\
\mathbb{x} ::= & f \mid \mathbb{t} & \text{extended value} \\
\mathbb{a} ::= & e \mid \nu c[se] \mid \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t} \mid \nu f\colon \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t} \mid f^{\checkmark} & \text{atom} \\
\mathbb{b} ::= & \mathbb{a} \triangleright \Gamma \mid \mathbb{a} \,⨟\, \mathbb{b} \mid (se)\{\, \mathbb{b} \,\} \mid \mathbb{b} + \mathbb{b} & \text{behavioural type}
\end{array}$$

where c, c', $\cdots$ range over cog names and f, f', $\cdots$ range over future names. Basic values $\mathbb{t}$ are either generic (non-size) expressions $-$ or size expressions se or the type $c[se]$ of an object of cog c with capacity se. The extended values add future names to basic values.

Atoms $\mathbb{a}$ define creation of cogs ($\nu c[se]$), synchronous and asynchronous method invocations ($\mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t}$ and $\nu f\colon \mathtt{m}(\overline{\mathbb{t}}) \to \mathbb{t}$, respectively), and synchronizations on asynchronous invocations ($f^{\checkmark}$). We observe that cog creations always carry a capacity, which has to be a size expression because our analysis in the next section cannot deal with generic expressions. Behavioural types $\mathbb{b}$ are sequences of atoms $\mathbb{a} \fatsemi \mathbb{b}'$ or conditionals, typically $(se)\{\ \mathbb{b}\ \} + (\neg se)\{\ \mathbb{b}'\ \}$ or $\mathbb{b} + \mathbb{b}'$, according to whether the boolean guard is a size expression that depends on the arguments of a method or not. In order to type sequential composition in a precise way (see rule T-Seq), the leaves of behavioural types are labelled with *environments*, ranged over by Γ, Γ', $\cdots$. Environments are maps from method names $\mathtt{m}$ to terms $(\overline{\mathbb{t}}) \to \mathbb{t}$, from variables to extended values $\mathbb{x}$, and from future names to values that are either $\mathbb{t}$ or $\mathbb{t}^{\checkmark}$.

The abstract behaviour of methods is defined by *method behavioural types* of the form: $\mathtt{m}(\mathbb{t}_t, \overline{\mathbb{t}})\{\ \mathbb{b}\ \} : \mathbb{t}_r$, where $\mathbb{t}_t$ is the type value of the receiver of the method, $\overline{\mathbb{t}}$ are the type value of the arguments, $\mathbb{b}$ is the abstract behaviour of the body, and $\mathbb{t}_r$ is the type value of the returned object. The subterm $\mathbb{t}_t, \overline{\mathbb{t}}$ of the method contract is called *header*; $\mathbb{t}_r$ is called *returned type value*. We assume that names in the header occur linearly. Names in the header *bind* the names in $\mathbb{b}$ and in $\mathbb{t}_r$. The header and the returned type value, written $(\mathbb{t}_t, \overline{\mathbb{t}}) \to \mathbb{t}_r$, are called *behavioural type signature*. Names occurring in $\mathbb{b}$ or $\mathbb{t}_r$ may be *not bound* by header. These *free names* correspond to new cog creations and will be replaced by fresh cog names during the analysis. We use $\mathbb{C}$ to range over method behavioural types.

The type system uses judgments of the following form:

- $\Gamma \vdash e : \mathbb{x}$ for pure expressions e, $\Gamma \vdash f : \mathbb{t}$ or $\Gamma \vdash f : \mathbb{t}^{\checkmark}$ for future names f, and $\Gamma \vdash \mathtt{m}(\overline{\mathbb{t}}) : \mathbb{t}$ for methods.
- $\Gamma \vdash z : \mathbb{x},\ [\![\mathbb{a} \rhd \Gamma']\!]$ for expressions with side effects z, where $\mathbb{x}$ is the value, $\mathbb{a} \rhd \Gamma'$ is the corresponding behavioural type, where Γ' is the environment Γ *with possible updates* of variables and future names.
- $\Gamma \vdash s : \mathbb{b}$, in this case the updated environments Γ' are inside the behavioural type, in correspondence of every branch of its.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. Moreover, we define

$$\Gamma[x \mapsto \mathbb{x}](y) \ \overset{def}{=} \ \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

The *multi-hole contexts* $\mathcal{C}[\ ]$ are defined by the following syntax:

$$\mathcal{C}[\ ] \ ::= \quad [\ ] \quad | \quad \mathbb{a} \fatsemi \mathcal{C}[\ ] \quad | \quad \mathcal{C}[\ ] + \mathcal{C}[\ ] \quad | \quad (se)\{\mathcal{C}[\ ]\}$$

and, whenever $\mathbb{b} = \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]$, then $\mathbb{b}[x \mapsto \mathbb{x}]$ is defined as $\mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1[x \mapsto \mathbb{x}]] \cdots [\mathbb{a}_n \rhd \Gamma_n[x \mapsto \mathbb{x}]]$.

The typing rules for expressions are defined in Figure 3. These rules are not standard because (size) expressions containing method's arguments are typed with the expressions themselves. This is crucial to the cost analysis in Section 5. In particular, *cog creation* is typed by rule T-New, with value $c[se]$, where c is the fresh name associated with the new cog and se is the value associated with the declared capacity. The behavioural type for the cog creation is $\nu c[se] \rhd \Gamma[c \mapsto se]$, where the newly created cog is added to Γ. In this way, it is possible to verify whether the receiver of a method invocation is within a locally created cog or not by testing whether the receiver belongs to $\mathrm{dom}(\Gamma)$ or not,

$$\text{(T-Var)}\ \dfrac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(T-Se)}\ \Gamma \vdash se : se \qquad \text{(T-Nse)}\ \Gamma \vdash nse : - \qquad \text{(T-Method)}\ \dfrac{\begin{array}{c}\Gamma(\mathtt{m}) = (\overline{\mathbb{t}}) \rightarrow \mathbb{t}' \\ fv(\mathbb{t}') \setminus fv(\overline{\mathbb{t}}) \neq \varnothing \ \text{ implies } \ \sigma(\mathbb{t}') \text{ fresh}\end{array}}{\Gamma \vdash \mathtt{m}(\sigma(\overline{\mathbb{t}})) : \sigma(\mathbb{t}')}$$

$$\text{(T-New)}\ \dfrac{\Gamma \vdash e : se \quad c \text{ fresh}}{\Gamma \vdash \mathtt{new\ Class\ with}\ e : c[se],\ \big[\nu c[se] \rhd \Gamma[c \mapsto se]\big]} \qquad \text{(T-New-Local)}\ \dfrac{\Gamma \vdash \mathtt{this} : c[se]}{\Gamma \vdash \mathtt{new\ local\ Class} : c[se],\ \big[\mathtt{0} \rhd \Gamma\big]}$$

$$\text{(T-Invoke-Sync)}\ \dfrac{\begin{array}{c}\Gamma \vdash e : c[se] \quad \Gamma(\mathtt{this}) = c[se] \\ \Gamma \vdash \overline{e} : \overline{\mathbb{t}} \quad \Gamma \vdash \mathtt{m}(c[se], \overline{\mathbb{t}}) : \mathbb{t}'\end{array}}{\Gamma \vdash e.\mathtt{m}(\overline{e}) : \mathbb{t}',\ \big[\mathtt{m}(c[se], \overline{\mathbb{t}}) \rightarrow \mathbb{t}' \rhd \Gamma\big]} \qquad \text{(T-Invoke)}\ \dfrac{\begin{array}{c}\Gamma \vdash e : c[se] \quad (c \in \mathrm{dom}(\Gamma) \ \text{ or } \ \Gamma(\mathtt{this}) = c[se]) \\ \Gamma \vdash \overline{e} : \overline{\mathbb{t}} \quad \Gamma \vdash \mathtt{m}(c[se], \overline{\mathbb{t}}) : \mathbb{t}' \quad f \text{ fresh}\end{array}}{\Gamma \vdash e\mathtt{!m}(\overline{e}) : f,\ \big[\nu f \colon \mathtt{m}(c[se], \overline{\mathbb{t}}) \rightarrow \mathbb{t}' \rhd \Gamma[f \mapsto \mathbb{t}']\big]}$$

$$\text{(T-Get)}\ \dfrac{\Gamma \vdash e : f \quad \Gamma(f) = \mathbb{t}}{\Gamma \vdash e.\mathtt{get} : \mathbb{t},\ \big[f^{\checkmark} \rhd \Gamma[f \mapsto \mathbb{t}^{\checkmark}]\big]} \qquad \text{(T-Get-Top)}\ \dfrac{\Gamma \vdash e : f \quad \Gamma(f) = \mathbb{t}^{\checkmark}}{\Gamma \vdash e.\mathtt{get} : \mathbb{t},\ \big[\mathtt{0} \rhd \Gamma\big]}$$

Fig. 3. Typing rules for expressions

respectively (*cf.* rule T-Invoke). *Object creation* (*cf.* rule T-New-Local) is typed as the cog creation, with the exception that the cog name and the capacity value are taken from the local cog and the behavioural type is empty. Rule T-Invoke types *method invocations* $e\mathtt{!m}(\overline{e})$ by using a fresh future name f that is associated to the method name, the cog name of the callee and the arguments. In the updated environment, f is associated with the returned value. Next we discuss the constraints in the premise of the rule. As we discussed in Section 2, asynchronous invocations are allowed on callees located in the current cog, $\Gamma(\mathtt{this}) = c[se]$, or on a newly created object which resides in a fresh cog, $c \in \mathrm{dom}(\Gamma)$. Rule T-Get defines the *synchronization* with a method invocation that corresponds to a future f. The expression is typed with the value $\mathbb{t}$ of f in the environment and behavioural type $f^{\checkmark}$. Γ is then updated for recording that the synchronization has been already performed, thus any subsequent synchronization on the same value would not imply any waiting time (see that in rule T-Get-Top the behavioural type is $\mathtt{0}$). The *synchronous method invocation* in rule T-Invoke-Sync is directly typed with the return value $\mathbb{t}'$ of the method and with the corresponding behavioural type. The rule enforces that the cog of the callee coincides with the local one.

The typing rules for statements are presented in Figure 4. The behavioural type in rule T-Job expresses the time consumption for an object with capacity se' to perform se processing cycles: this time is given by se/se', which we observe is in general a rational number. We will return to this point in Section 5.

The typing rules for method and class declarations are shown in Figure 5.

(T-Assign)
$$\frac{\Gamma \vdash \mathit{rhs} : \mathbb{x}, \ \left[\mathbb{a} \rhd \Gamma'\right]}{\Gamma \vdash x = \mathit{rhs} : \mathbb{a} \rhd \Gamma'[x \mapsto \mathbb{x}]}$$

(T-Job)
$$\frac{\Gamma \vdash e : \mathit{se} \quad \Gamma \vdash \texttt{this} : c[\mathit{se}']}{\Gamma \vdash \texttt{job}(e) : \mathit{se}/\mathit{se}' \rhd \Gamma}$$

(T-Return)
$$\frac{\Gamma \vdash e : \mathbb{t} \quad \Gamma \vdash \texttt{destiny} : \mathbb{t}}{\Gamma \vdash \texttt{return}\ e : 0 \rhd \Gamma}$$

(T-Seq)
$$\frac{\begin{array}{c}\Gamma \vdash s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n] \\ \Gamma_i \vdash s' : \mathbb{b}'_i\end{array}}{\Gamma \vdash s\ \texttt{;}\ s' : \mathcal{C}[\mathbb{a}_1 \fatsemi \mathbb{b}'_1] \cdots [\mathbb{a}_n \fatsemi \mathbb{b}'_n]}$$

(T-If-Nse)
$$\frac{\Gamma \vdash e : - \quad \Gamma \vdash s : \mathbb{b} \quad \Gamma \vdash s' : \mathbb{b}'}{\Gamma \vdash \texttt{if}\ e\ \{\, s \,\}\ \texttt{else}\ \{\, s' \,\} : \mathbb{b} + \mathbb{b}'}$$

(T-If-Se)
$$\frac{\Gamma \vdash e : \mathit{se} \quad \Gamma \vdash s : \mathbb{b} \quad \Gamma \vdash s' : \mathbb{b}'}{\Gamma \vdash \texttt{if}\ e\ \{\, s \,\}\ \texttt{else}\ \{\, s' \,\} : (\mathit{se})\{\mathbb{b}\} + (\neg \mathit{se})\{\mathbb{b}'\}}$$

Fig. 4. Typing rules for statements

(T-Method)
$$\frac{\begin{array}{c}\Gamma(\texttt{m}) = (\mathbb{t}_t, \overline{\mathbb{t}}) \to \mathbb{t}_r \\ \Gamma[\texttt{this} \mapsto \mathbb{t}_t][\texttt{destiny} \mapsto \mathbb{t}_r][\overline{x} \mapsto \overline{\mathbb{t}}] \vdash s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]\end{array}}{\Gamma \vdash T\ \texttt{m}\ (\overline{T\ x})\ \{\, s \,\} : \texttt{m}(\mathbb{t}_t, \overline{\mathbb{t}})\{\, \mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing] \,\} : \mathbb{t}_r}$$

(T-Class)
$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}} \quad \Gamma[\texttt{this} \mapsto \mathrm{start}[k]][\overline{x} \mapsto \overline{\mathbb{t}}] \vdash s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]}{\Gamma \vdash \overline{M}\ \{\, \overline{T\ x\ \texttt{;}}\ s \,\}\ \texttt{with}\ k\ : \overline{\mathbb{C}}, \mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing]}$$

Fig. 5. Typing rules for declarations

Examples The behavioural type of the `fib` method discussed in Section 2 is

```
fib(c[x],n) {
   (n ≤ 1){ 0▷∅ }
 + (n ≥ 2){
    1/x ⨟ d[x] ⨟ νf: fib(c[x],n-1)→ - ⨟ νg: fib(d[x],n-2)→ - ⨟
    f✓⨟ g✓⨟0▷∅ } } : -
```

5 The time analysis

The behavioural types returned by the system defined in Section 4 are used to compute upper bounds of time complexity of a `tml` program. This computation is performed by an off-the-shelf solver – the `CoFloCo` solver [4] – and, in this section, we discuss the translation of a behavioural type program into a set of *cost equations* that are fed to the solver. These cost equations are terms

$$m(\overline{x}) = \mathit{exp} \quad [\mathit{se}]$$

where m is a (cost) function symbol, *exp* is an expression that may contain (cost) function symbol applications (we do not define the syntax of *exp*, which may be derived by the following equations; the reader may refer to [4]), and *se* is a size expression whose variables are contained in $\overline{x}$. Basically, our translation maps method types into cost equations, where (i) method invocations are translated into function applications, and (ii) cost expressions *se* occurring in the types are left unmodified. The difficulties of the translation is that the cost equations must account for the parallelism of processes in different cogs and for sequentiality of processes in the same cog. For example, in the following code:

```
x = new Class with c; y = new Class with d;
f = x!m(); g = y!n(); u = g.get; u = f.get;
```

the invocations of `m` and `n` will run in parallel, therefore their cost will be $\mathtt{max}(t, t')$, where t is the time of executing `m` on `x` and t' is the time executing `n` on `y`. On the contrary, in the code

```
x = new local Class; y = new local Class;
f = x!m(); g = y!n(); u = g.get; u = f.get;
```

the two invocations are queued for being executed on the same cog. Therefore the time needed for executing them will be $t + t'$, where t is time needed for executing `m` on `x`, and t' is the time needed for executing `n` on `y`. To abstract away the execution order of the invocations, the execution time of *all unsynchronized* methods from the same cog are taken into account when one of these methods is synchronized with a `get`-statement. To avoid calculating the execution time of the rest of the unsynchronized methods in the same cog more than necessary, their estimated cost are ignored when they are later synchronized.

In this example, when the method invocation `y!n()` is synchronized with `g.get`, the estimated time taken is $t+t'$, which is the sum of the execution time of the two unsynchronized invocations, including the time taken for executing `m` on `x` because both `x` and `y` are residing in the same cog. Later when synchronizing the method invocation `x!m()`, the cost is considered to be *zero* because this invocation has been taken into account earlier.

The translate function. The translation of behavioural types into cost equations is carried out by the function `translate`, defined below. This function parses atoms, behavioural types or declarations of methods and classes. We will use the following auxiliary function that removes cog names from (tuples of) $\mathbb{t}$ terms:

$$\lfloor \_ \rfloor = \_ \qquad \lfloor e \rfloor = e \qquad \lfloor c[e] \rfloor = e \qquad \lfloor \mathbb{t}_1, \ldots, \mathbb{t}_n \rfloor = \lfloor \mathbb{t}_1 \rfloor, \ldots, \lfloor \mathbb{t}_n \rfloor$$

We will also use *translation environments*, ranged over by Ψ, Ψ', $\cdots$, which map future names to pairs $(e, \mathtt{m}(\overline{\mathbb{t}}))$ that records the (over-approximation of the) time when the method has been invoked and the invocation.

In the case of atoms, `translate` takes four inputs: a *translation environment* Ψ, the cog name of the carrier, an over-approximated cost e of an execution branch, and the atom a. In this case, `translate` returns an updated translation environment and the cost. It is defined as follows.

$$
\texttt{translate}(\Psi, c, e, \mathbb{a}) \;=\;
\begin{cases}
(\Psi, e + e') & \text{when } \mathbb{a} = e' \\
(\Psi, e) & \text{when } \mathbb{a} = \nu c[e'] \\
(\Psi, e + \texttt{m}(\lfloor\overline{\mathbb{t}}\rfloor)) & \text{when } \mathbb{a} = \texttt{m}(\overline{\mathbb{t}}) \to \mathbb{t}' \\
(\Psi[f \mapsto (e, \texttt{m}(\overline{\mathbb{t}}))],\, e) & \text{when } \mathbb{a} = (\nu f \colon \texttt{m}(\overline{\mathbb{t}}) \to \mathbb{t}') \\
(\Psi \setminus F, e + e_1))) & \text{when } \mathbb{a} = f^{\checkmark} \quad \text{and} \quad \Psi(f) = (e_f, \texttt{m}_f(c[e'], \overline{\mathbb{t}_f})) \\
 & \text{let } F = \{\, g \mid \Psi(g) = (e_g,\, \texttt{m}_g(c[e'], \overline{\mathbb{t}_g})) \,\} \text{ then} \\
 & \text{and } e_1 = \sum \{\, \texttt{m}_g(\lfloor\overline{\mathbb{t}'_g}\rfloor) \mid (e_g, \texttt{m}_g(\overline{\mathbb{t}'_g})) \in \Psi(F) \,\} \\
(\Psi \setminus F, \mathit{max}(e, e_1 + e_2)) & \text{when } \mathbb{a} = f^{\checkmark} \text{ and } \Psi(f) = (e_f, \texttt{m}_f(c'[e'], \overline{\mathbb{t}_f})) \text{ and } c \neq c' \\
 & \text{let } F = \{\, g \mid \Psi(g) = (e_g,\, \texttt{m}_g(c'[e'], \overline{\mathbb{t}_g})) \,\} \text{ then} \\
 & e_1 = \sum \{\, \texttt{m}_g(\lfloor\overline{\mathbb{t}'_g}\rfloor) \mid (e_g, \texttt{m}_g(\overline{\mathbb{t}'_g})) \in \Psi(F) \,\} \\
 & \text{and } e_2 = \mathit{max}\{\, e_g \mid (e_g, \texttt{m}_g(\overline{\mathbb{t}'_g})) \in \Psi(F) \,\} \\
(\Psi, e) & \text{when } \mathbb{a} = f^{\checkmark} \text{ and } f \notin \mathrm{dom}(\Psi)
\end{cases}
$$

The interesting case of `translate` is when the atom is $f^{\checkmark}$. There are three cases:

1. The synchronization is with a method whose callee is an object of the same cog. In this case its cost must be *added*. However, it is not possible to know when the method will be actually scheduled. Therefore, we sum the costs of all the methods running on the same cog (worst case) – the set F in the formula – and we remove them from the translation environment.
2. The synchronization is with a method whose callee is an object on a different cog c'. In this case we use the cost that we stored in $\Psi(f)$. Let $\Psi(f) = (e_f, \texttt{m}_f(c'[e'], \overline{\mathbb{t}_f}))$, then e_f represents the time of the invocation. The cost of the invocation is therefore $e_f + \texttt{m}_f(e', \lfloor\overline{\mathbb{t}_f}\rfloor)$. Since the invocation is *in parallel* with the thread of the cog c, the overall cost is $\mathit{max}(e, e_f + \texttt{m}_f(e', \lfloor\overline{\mathbb{t}_f}\rfloor))$. As in case 1, we consider the worst scheduler choice on c'. Instead of taking $e_f + \texttt{m}_f(e', \lfloor\overline{\mathbb{t}_f}\rfloor)$, we compute the cost of all the methods running on c' – the set F in the formula – and we remove them from the translation environment.
3. The future does not belong to Ψ. That is the cost of the invocation which has been already computed. In this case, the value e does not change.

In the case of behavioural types, `translate` takes as input a translation environment, the cog name of the carrier, an over-approximated cost of the current execution branch $(e_1)e_2$, where e_1 indicates the conditions corresponding to the branch, and the behavioural type $\mathbb{a}$.

$$
\texttt{translate}(\Psi, c, (e_1)e_2, \mathbb{b}) \;=\;
\begin{cases}
\{\, (\Psi', (e_1)e'_2) \,\} & \text{when } \mathbb{b} = \mathbb{a} \triangleright \Gamma \quad \text{and} \quad \texttt{translate}(\Psi, c, e_2, \mathbb{a}) = (\Psi', e'_2) \\
C & \text{when } \mathbb{b} = \mathbb{a} \fatsemi \mathbb{b}' \quad \text{and} \quad \texttt{translate}(\Psi, c, e_2, \mathbb{a}) = (\Psi', e'_2) \\
 & \text{and } \texttt{translate}(\Psi', c, (e_1)e'_2, \mathbb{b}') = C \\
C \cup C' & \text{when } \mathbb{b} = \mathbb{b}_1 + \mathbb{b}_2 \quad \text{and} \quad \texttt{translate}(\Psi, c, (e_1)e_2, \mathbb{b}_1) = C \\
 & \text{and} \quad \texttt{translate}(\Psi, c, (e_1)e_2, \mathbb{b}_2) = C' \\
C & \text{when } \mathbb{b} = (e)\{\, \mathbb{b}' \,\} \quad \text{and} \quad \texttt{translate}(\Psi, c, (e_1 \wedge e)e_2, \mathbb{b}') = C
\end{cases}
$$

The translation of the behavioural types of a method is given below. Let $\mathrm{dom}(\Psi) = \{\, f_1, \cdots, f_n \,\}$. Then we define $\Psi^{\checkmark} \overset{def}{=} {f_1}^{\checkmark} \fatsemi \cdots \fatsemi {f_n}^{\checkmark}$.

$$\texttt{translate}(\texttt{m}(c[e], \overline{\mathbb{t}})\{\, \mathbb{b} \,\} : \mathbb{t}) \;=\; \left[\begin{array}{ll} \texttt{m}(e, \overline{e}) = e'_1 + e''_1 & [e_1] \\ \quad\vdots & \\ \texttt{m}(e, \overline{e}) = e'_n + e''_n & [e_n] \end{array}\right.$$

where $\texttt{translate}(\varnothing, c, 0, \mathbb{b}) = \{\, \Psi_i, (e_i)e'_i \mid 1 \leq i \leq n \,\}$, and $\overline{e} = \lfloor \overline{\mathbb{t}} \rfloor$, and $e''_i = \texttt{translate}(\Psi_i, c, 0, \Psi_i^{\checkmark} \rhd \varnothing)$. In addition, $[e_i]$ are the conditions for branching the possible execution paths of method $\texttt{m}(e, \overline{e})$, and $e'_i + e''_i$ is the over-approximation of the cost for each path. In particular, e'_i corresponds to the cost of the synchronized operations in each path (e.g., `jobs` and `gets`), while e''_i corresponds to the cost of the asynchronous method invocations triggered by the method, but not synchronized within the method body.

Examples We show the translation of the behavioural type of fibonacci presented in Section 4. Let $\mathbb{b} = (se)\{0 \rhd \varnothing\} + (\neg se)\{\mathbb{b}'\}$, where $se = (\mathrm{n} \leq 1)$ and $\mathbb{b}' = 1/e \,⨾\, \nu f\colon \texttt{fib}(c[e], n-1) \to - \,⨾\, \nu g\colon \texttt{fib}(c'[e], n-2) \to - \,⨾\, f^{\checkmark} \,⨾\, g^{\checkmark} \,⨾\, 0 \rhd \varnothing\}$. Let also $\Psi = \Psi_1 \cup \Psi_2$, where $\Psi_1 = [f \mapsto (1/e, \texttt{fib}(e, n-1))]$ and $\Psi_2 = [g \mapsto (1/e, \texttt{fib}(e, n-2))]$.

The following equations summarize the translation of the behavioural type of the fibonacci method.

$$\begin{array}{l}
\texttt{translate}(\varnothing, c, 0, \mathbb{b}) \\
\quad = \texttt{translate}(\varnothing, c, 0, (se)\ \{\, 0 \rhd \varnothing \,\}) \;\cup\; \texttt{translate}(\varnothing, c, 0, (\neg se)\ \{\, \mathbb{b}' \,\}) \\
\quad = \texttt{translate}(\varnothing, c, (se)0, \{\, 0 \rhd \varnothing \,\}) \;\cup\; \texttt{translate}(\varnothing, c, (\neg se)0, \{\, 1/e \ ⨾\ \ldots \,\}) \\
\quad = \{\, (se)0 \,\} \;\cup\; \texttt{translate}(\varnothing, c, (\neg se)(1/e), \{\, \nu f\colon \texttt{fib}(c[e], n-1) \to - \ ⨾ \ldots \,\}) \\
\quad = \{\, (se)0 \,\} \;\cup\; \texttt{translate}(\Psi_1, c, (\neg se)(1/e), \{\, \nu g\colon \texttt{fib}(c'[e], n-2) \to - \ ⨾ \ldots \,\}) \\
\quad = \{\, (se)0 \,\} \;\cup\; \texttt{translate}(\Psi, c, (\neg se)(1/e), \{\, f^{\checkmark} ⨾ g^{\checkmark} ⨾ \ldots \,\}) \\
\quad = \{\, (se)0 \,\} \;\cup\; \texttt{translate}(\Psi_2, c, (\neg se)(1/e + \texttt{fib}(e, n-1)), \{\, g^{\checkmark} ⨾ \ldots \,\}) \\
\quad = \{\, (se)0 \,\} \;\cup\; \texttt{translate}(\varnothing, c, (\neg se)(1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2))), \{\, 0 \rhd \varnothing \,\}) \\
\quad = \{\, (se)0 \,\} \;\cup\; \{\, (\neg se)(1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2))) \,\}
\end{array}$$

$$\begin{array}{l}
\texttt{translate}(\varnothing, c, 0, 0) \;=\; (\varnothing, 0) \\
\texttt{translate}(\varnothing, c, 0, 1/e) \;=\; (\varnothing, 1/e) \\
\texttt{translate}(\varnothing, c, 1/e, \nu f\colon \texttt{fib}(c[e], n-1) \to - \,) \;=\; (\Psi_1, 1/e) \\
\texttt{translate}(\Psi_1, c, 1/e, \nu g\colon \texttt{fib}(c'[e], n-2) \to - \,) \;=\; (\Psi, 1/e) \\
\texttt{translate}(\Psi, c, 1/e, f^{\checkmark}) \;=\; (\Psi_2, 1/e + \texttt{fib}(e, n-1)) \\
\texttt{translate}(\Psi_2, c, 1/e + \texttt{fib}(e, n-1), g^{\checkmark}) \;=\; (\varnothing, 1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2)))
\end{array}$$

$$\begin{array}{l}
\texttt{translate}(\texttt{fib}\ (c[e], n)\{\, \mathbb{b} \,\} : -) \;= \\
\begin{cases} \texttt{fib}(e, n) = 0 & [n\ \leq 1] \\ & \\ \texttt{fib}(e, n) = 1/e + \max(\texttt{fib}(e, n-1), \texttt{fib}(e, n-2)) & [n\ \geq 2] \end{cases}
\end{array}$$

Remark 1. Rational numbers are produced by the rule T-Job of our type system. In particular behavioural types may manifest terms se/se' where se gives the processing cycles defined by the `job` operation and se' specifies the number of processing cycles per unit of time the corresponding cog is able to handle. Unfortunately, our backend solver – `CoFloCo` – cannot handle rationals se/se'

where se' is a variable. This is the case, for instance, of our fibonacci example, where the cost of each iteration is `1/x`, where `x` is a parameter. In order to analyse this example, we need to determine *a priori* the capacity to be a constant – say 2 –, obtaining the following input for the solver:

```
eq(f(E,N),0,[],[-N>=1,2*E=1]).
eq(f(E,N),nat(E),[f(E,N-1)],[N>=2,2*E=1]).
eq(f(E,N),nat(E),[f(E,N-2)],[N>=2,2*E=1]).
```

Then the solver gives `nat(N-1)*(1/2)` as the upper bound. It is worth to notice that fixing the fibonacci method is easy because the capacity does not change during the evaluation of the method. This is not always the case, as in the following alternative definition of fibonacci:

```
Int fib_alt(Int n) {
    if (n<=1) { return 1; }
    else { Fut<Int> f; Class z; Int m1; Int m2;
           job(1);
           z = new Class with (this.capacity*2) ;
           f = this!fib_alt(n-1); g = z!fib_alt(n-2);
           m1 = f.get; m2 = g.get;
           return m1+m2; } }
```

In this case, the recursive invocation `z!fib_alt(n-2)` is performed on a cog with twice the capacity of the current one and `CoFloCo` is not able to handle it. It is worth to observe that this is a problem of the solver, which is otherwise very powerful for most of the examples. Our behavioural types carry enough information for dealing with more complex examples, so we will consider alternative solvers or combination of them for dealing with examples like `fib_alt`.

6 Properties

In order to prove the correctness of our system, we need to show that (*i*) the behavioural type system is correct, and (*ii*) the computation time returned by the solver is an upper bound of the actual cost of the computation.

The correctness of the type system in Section 4 is demonstrated by means of a subject reduction theorem expressing that if a runtime configuration cn is well typed and $cn \to cn'$ then cn' is well-typed as well, and the computation time of cn is larger or equal to that of cn'. In order to formalize this theorem we extend the typing to configurations and we also use extended behavioural types $\mathbb{k}$ with the following syntax

$$\mathbb{k} ::= \quad \mathbb{b} \ \mid \ [\mathbb{b}]^{c}_{f} \ \mid \ \mathbb{k} \parallel \mathbb{k} \qquad\qquad \text{runtime behavioural type}$$

The type $[\mathbb{b}]^{c}_{f}$ expresses the behaviour of an asynchronous method bound to the future f and running in the cog c; the type $\mathbb{k} \parallel \mathbb{k}'$ expresses the parallel execution of methods in $\mathbb{k}$ and in $\mathbb{k}'$.

We then define a relation $\trianglerighteq_t$ between runtime behavioural types that relates types. The definition is algebraic, and $\mathbb{k} \trianglerighteq_t \mathbb{k}'$ is intended to mean that the computational time of $\mathbb{k}$ is at least that of $\mathbb{k}'+t$ (or conversely the computational time of $\mathbb{k}'$ is at most that of $\mathbb{k}-t$). This is actually the purpose of our theorems.

Theorem 1 (Subject Reduction). *Let cn be a configuration of a* `tml` *program and let $\Bbbk$ be its behavioural type. If cn is not strongly t-stable and $cn \to cn'$ then there exists $\Bbbk'$ typing cn' such that $\Bbbk \trianglerighteq_0 \Bbbk'$. If cn is strongly t-stable and $cn \to cn'$ then there exists $\Bbbk'$ typing cn' such that $\Bbbk \trianglerighteq_t \Bbbk'$.*

The proof of is a standard case analysis on the last reduction rule applied. The second part of the proof requires an extension of the `translate` function to runtime behavioural types. We therefore define a cost of the equations $\mathcal{E}_{\Bbbk}$ returned by $\texttt{translate}(\Bbbk)$ – noted $\texttt{cost}(\mathcal{E}_{\Bbbk})$ – by unfolding the equational definitions.

Theorem 2 (Correctness). *If $\Bbbk \trianglerighteq_t \Bbbk'$, then $\texttt{cost}(\mathcal{E}_{\Bbbk}) \ \geq \ \texttt{cost}(\mathcal{E}_{\Bbbk'}) + t$.*

As a byproduct of Theorems 1 and 2, we obtain the correctness of our technique, modulo the correctness of the solver.

7 Related work

In contrast to the static time analysis for sequential executions proposed in [7], the paper proposes an approach to analyse time complexity for concurrent programs. Instead of using a Hoare-style proof system to reason about end-user deadlines, we estimate the execution time of a concurrent program by deriving the time-consuming behaviour with a type-and-effect system.

Static time analysis approaches for concurrent programs can be divided into two main categories: those based on type-and-effect systems and those based on abstract interpretation – see references in [9]. Type-and-effect systems (i) collect constraints on type and resource variables and (ii) solve these constraints. The difference with respect to our approach is that we do not perform the analysis during the type inference. We use the type system for deriving behavioural types of methods and, in a second phase, we use them to run a (non compositional) analysis that returns cost upper bounds. This dichotomy allows us to be more precise, avoiding unification of variables that are performed during the type derivation. In addition, we notice that the techniques in the literature are devised for programs where parallel modules of sequential code are running. The concurrency is not part of the language, but used for parallelising the execution.

Abstract interpretation techniques have been proposed addressing domains carrying quantitative information, such as resource consumption. One of the main advantages of abstract interpretation is the fact that many practically useful optimization techniques have been developed for it. Consequently, several well-developed automatic solvers for cost analysis already exist. These techniques either use finite domains or use expedients (widening or narrowing functions) to guarantee the termination of the fix-point generation. For this reason, solvers often return inaccurate answers when fed with systems that are finite but not statically bounded. For instance, an abstract interpretation technique that is very close to our contribution is [2]. The analysis of this paper targets a language with the same concurrency model as ours, and the backend solver for our analysis, `CoFloCo`, is a slightly modified version of the solver used by [2]. However the two techniques differ profoundly in the resulting cost equations and in the way they are produced. Our technique computes the cost by means of a type system, therefore every method has an associated type, which is parametric with respect to the arguments. Then these types are translated into a bunch of cost equations that may be *composed* with those of other methods. So our approach supports a technique similar to *separate compilation*, and is able to deal with systems that create statically an unbounded but finite number of nodes. On the contrary, the technique in [2] is not compositional because it takes the whole program and computes the parts that may run in parallel. Then the cost equations are generated accordingly. This has the advantage that their technique does not have any restriction on invocations on arguments of methods that are (currently) present in our one.

We finally observe that our behavioural types may play a relevant role in a cloud computing setting because they may be considered as abstract descriptions of a method suited for SLA compliance.

8 Conclusions

This article presents a technique for computing the time of concurrent object-oriented programs by using behavioural types. The programming language we have studied features an explicit cost annotation operation that define the number of machine cycles required before executing the continuation. The actual computation activities of the program are abstracted by `job`-statements, which are the unique operations that consume time. The computational cost is then measured by introducing the notion of (strong) *t-stability* (*cf.* Definition 1), which represents the ticking of time and expresses that up to t time steps no control activities are possible. A Subject Reduction theorem (Theorem 1), then, relates this stability property to the derived types by stating that the consumption of t time steps by `job` statements is properly reflected in the type system. Finally, Theorem 2 states that the solution of the cost equations obtained by translation of the types provides an upper bound of the execution times provided by the type system and thus, by Theorem 1, of the actual computational cost.

Our behavioural types are translated into so-called cost equations that are fed to a solver that is already available in the literature – the `CoFloCo` solver [4]. As discussed in Remark 1, `CoFloCo` cannot handle rational numbers with variables at the denominator. In our system, this happens very often. In fact, the number pc of processing cycles needed for the computation of a `job`(pc) is divided by the speed s of the machine running it. This gives the cost in terms of time of the `job`(pc) statement. When the capacity is not a constant, but depends on the value of some parameter and changes over time, then we get the untreatable rational expression. It is worth to observe that this is a problem of the solver (otherwise very powerful for most of the examples), while our behavioural types carry enough information for computing the cost also in these cases. We plan to consider alternative solvers or a combination of them for dealing with complex examples.

Our current technique does not address the full language. In particular we are still not able to compute costs of methods that contain invocations to arguments which do not live in the same machine (which is formalized by the notion of cog in our language). In fact, in this case it is not possible to estimate the cost without any indication of the state of the remote machine. A possible solution to this issue is to deliver costs of methods that are parametric with respect to the state of remote machines passed as argument. We will investigate this solution in future work.

In this paper, the cost of a method also includes the cost of the asynchronous invocations in its body that have not been synchronized. A more refined analysis, combined with the resource analysis of [5], might consider the cost of each machine, instead of the overall cost. That is, one should count the cost of a method *per* machine rather than in a cumulative way. While these values are identical when the invocations are always synchronized, this is not the case for unsynchronized invocation and a disaggregated analysis might return better estimations of virtual machine usage.

References

1. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, 2014.
2. E. Albert, J. C. Fernández, E. B. Johnsen, and G. Román-Díez. Parallel cost analysis of distributed systems. In *Proceedings of SAS 2015*, volume 9291 of *Lecture Notes in Computer Science*. Springer, 2015. To appear.
3. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Sys.*, 25(6):599–616, 2009.
4. A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of 12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.
5. A. Garcia, C. Laneve, and M. Lienhardt. Static analysis of cloud elasticity. In *Proceedings of PPDP 2015*, 2015.
6. R. Hähnle and E. B. Johnsen. Resource-aware applications for the cloud. *IEEE Computer*, 48(6):72–75, 2015.
7. E. B. Johnsen, K. I. Pun, M. Steffen, S. L. Tapia Tarifa, and I. C. Yu. Meeting deadlines, elastically. In *From Action Systems to Distributed Systems: the Refinement Approach*. CRC Press, 2015. To Appear.
8. E. B. Johnsen, R. Schlatte, and S. L. 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.
9. P. W. Trinder, M. I. Cole, K. Hammond, H. Loidl, and G. Michaelson. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience*, 25(3):309–348, 2013.