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

Deliverable D2.2.1
Formalization of Service Contracts and SLAs (Initial Report)

Date of document: T18

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

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

Final version

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

Executive Summary:
Formalization of Service Contracts and SLAs (Initial Report)

This document summarises deliverable D2.2.1 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 contains the initial outcome of Task T2.2: basic formalizations of SLA and QoS interfaces and of SLAs addressing Task T2.2 goals

(i) develop a formal language for modeling SLA documents,

(ii) supplement behavioral interfaces with quality of services descriptions that address virtualized resources and deployment models.

List of Authors

Elena Giachino (BOL)
Cosimo Laneve (BOL)
Behrooz Nobakht (FRH)
Domenico Presenza (ENG)
Ka I Pun (UIO)
Rudolf Schlatte (UIO)
Amund Tveit (ATB)

Contents

Chapter 1

Introduction

According to the Envisage Description of Work, the Deliverable 2.2.1 contains the initial outcome towards the following two goals:

(i) develop a formal language for modeling Service Level Agreement (SLA) documents,

(ii) supplement behavioral interfaces with quality of services descriptions that address virtualized resources and deployment models.

SLAs define the contractual relationship between a cloud service customer and a provider. SLAs have a global nature, include legal requirements under different jurisdictions, have no standardisation of their format nor terminology, and do not abide by any precise definition. They are just a legal contract agreed upon by two parties. In order to be able to evaluate cloud services with respect to a SLA within Envisage, we need to have a way to specify SLA requirements in a formal way.

In Chapter 2 we present a formalisation of SLA, by means of the definition of suitable metrics which arise from case studies and industrial experience. SLA's metrics are formalized in terms of functions that depend on the resources allocated to a service and on the time windows. Some of the metrics identified in Chapter 2 have an intrinsic dynamic component which is not foreseeable to be verified statically (*e.g.* requests per minute) because they do not depend solely on the code but also on external intervention (*e.g.* the final service consumer) or on events in the environment (*e.g.* failures). Other metrics, on the contrary, can be measured by analysing the code (*e.g.* response time or resource capacity).

The scientific work on this task T2.2 addresses the statically verifiable metrics. We propose extensions of the behavioural interfaces studied in T2.1 (*c.f.* Deliverable 2.1) with information about response time and resource consumption, and we develop techniques for formally assessing the relationships between interfaces and virtualized services' software.

In Chapter 3 we define behavioural interfaces for the time analysis by following a *design by contract* methodology [17] for SLA-aware virtualized services. The methodology incorporates SLA requirements in the interfaces at the application level to ensure the QoS expectations of clients. In particular, the presented interface language specifies services, including their service contracts in form of response time guarantees – thus enabling the measurement of the Response Time metric (see Chapter 2). The target language is an extension of ABS that associates deadlines with method calls by means of clauses given in a style akin to JML [10] and Fresco [20] In Chapter 3 we discuss how to apply deductive verification techniques to ensure that all local deadlines are met during the execution of a virtualized service. At this stage the technique is restricted to sequential computation and synchronous method calls.

In Chapter 4 we define behavioural interfaces for the analysis of resources. Such interfaces take the form of behavioural types, which highlight the relevant operations affecting resource usage, such as creations, releases and concurrency. In Chapter 4 we address resources that are virtual machines; these resources are also called Deployment Components in Deliverables of tasks T1.2 and T1.3. A type system allows us to formalize the

association of behavioural types with programs and a translation function is used to derive inputs for a cost analysis tool that is being developed by TUD – the CoFloCo tool [11] – or by a tool developed at UCM – the PUBS tool [3]. The proof technique for demonstrating the correctness of the proposed approach for resource analysis is similar to the one developed in [12] for deadlock analysis. In [12], reported in Chapter 4 of Deliverable 2.1, we undertake a preliminary study of a type-based technique for analysing the deployment of resources in ABS. In particular, the language of [12] supported the creation of resources and their migration from one virtual machine to another. However, the migration resources is not of primary relevance in a cloud computing environment, where resources, once created and allocated to a virtual machine, do not move. Rather, in cloud computing, a full-fledged operation of release of resource is much more relevant and useful. As a consequence of this remark, in Chapter 4 we deliver a revision of the types and the technique in [12]. It is also with to observe that the behavioural types defined in Chapter 4 can be integrated in the ABS programs in a similar way as done in Chapter 3, using clauses given in a JML style [10]. That is, in principle, the two techniques of Chapters 3 and 4 may be combined in order to support both time and resource analyses at the same time.

The techniques developed in this Deliverable target extensions of ABS with operations for modelling time and resources. The definition of these extensions is part of WP1 and is discussed in Deliverable of T1.2.

1.1 List of Papers Comprising Deliverable D2.2.1

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"). Direct links are also provided for each paper listed below.

Paper 1: Meeting Deadlines, Elastically This paper presents a formal approach to modelling and verifying programs with response time guarantee, which is a non-functional property of virtualised services. We extends JML-like interfaces with response time annotations to model services written in a sequential object-oriented language. A Hoare-style proof system is developed to reason about the response time guarantees of services.

The paper was written by Einar Broch Johnsen, Ka I Pun, Martin Steffen, S. Lizeth Tapia Tarifa, and Ingrid Chieh Yu. The paper is accepted and will appear in the book *"From Action Systems to Distributed Systems: the Refinement Approach"*.

Download the paper at `http://www.ifi.uio.no/~violet/papers/kaisa.pdf`.

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 behavioral 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.

Download the paper at `http://www.cs.unibo.it/~laneve/papers/VM.pdf`.

Chapter 2

Formalisation of SLA metrics

2.1 QoS Metrics Catalog

In the "Cloud Service Level Agreement Standardisation Guidelines" document [13], cloud services are evaluated according to different Service Level Objectives, which are often associated with metrics. Metrics are defined measurement methods and measurement scales, and are used to set the boundaries and margins of errors that apply to the behaviour of the cloud service. Metrics may also be used at runtime for service monitoring, balancing, or remediation, as well as at static time to evaluate the code. The analysis of the metrics related to the code will be the focus of this chapter. In particular, we discuss metrics related to performance, such as *availability*, *response time*, and *capacity*.

Availability. Availability is the property of being accessible and usable upon demand and can be evaluated with respect to the following objectives. For every property, we give an informal description and we define whether the associated metric verification can be done statically (by means of behavioural interfaces) or dynamically (by means of runtime monitoring).

Property	**Description**	**Associated metric verification**
Level of uptime	refers to the time in a defined period the service was available, over the total possible available time, expressed as a percentage	runtime monitoring
Percentage of successful requests	refers to the number of requests processed by the service without an error over the total number of submitted requests, expressed as a percentage	runtime monitoring
Percentage of timely service provisioning requests	refers to the number of service provisioning requests completed within a defined time period over the total number of service provisioning requests, expressed as a percentage	runtime monitoring

Response Time. Response time is the time interval between a cloud service customer event and a cloud service provider response event, and can be evaluated with respect to the following properties.

Property	Description	Associated metric verification
Average response time	refers to the statistical mean over a set of cloud service response time observations for a particular form of request.	static via behavioural interfaces
Maximum response time	refers to the maximum response time target for a given particular form of request.	static via behavioural interfaces

Capacity. Capacity is the maximum amount of some property of a cloud service and can be evaluated with respect to the following properties.

Property	Description	Associated metric verification
Number of simultaneous connections	refers to the maximum number of separate connections to the cloud service at one time.	runtime monitoring
Number of simultaneous cloud service users	refers to a target for the maximum number of separate cloud service customer users that can be using the cloud service at one time.	runtime monitoring
Maximum resource capacity	refers to the maximum amount of a given resource available to an instance of the cloud service for a particular cloud service customer. Example resources include data storage, memory, number of CPU cores.	static via behavioural interfaces
Service Throughput	refers to the minimum number of specified requests that can be processed by the cloud service in a stated time period. (e.g. requests per minute).	runtime monitoring

Next we identify the metrics that we want to formalize and to verify. Some of these metrics will be verified statically, by applying one of the techniques described in Chapters 3 and 4 directly on the code. Other metrics are not verifiable by inspecting the code, but monitoring it with ad-hoc code – this information has been already added in the foregoing tables. Samples of the metrics are given below:

Statically verifiable metrics:
- *Average response time* is used to specify facts such as "Service takes 180 secs. to serve a request". A static analysis, as the one defined in Chapter 3, evaluates the time needed by every possible execution path and return the maximum value.
- *Consumed Resources* is used to specify facts such as "A service uses 4 virtual machines", and can be used to compute an upper bound of the total amount of resources needed by a service. The static analysis defined Chapter 4 allows one to estimate these upper bounds.
- *Service data size* is used to specify facts such as "The service transmits data which are 3 times the size of the input data" which can be used to estimate the total bandwidth needed by the service. This information can be returned by a static analysis computing the sizes of data manipulated by a program.

Dynamically verifiable metrics:
- *Service rate* is used to specify facts such as "The service processes 2000 service requests in a day". This is used to evaluate the conformance with respect to the Service Throughput.
- *Mean time between failures* is used to specify facts such as "Service is unavailable every 300 days", which is used for evaluating the Level of Uptime.
- *Recovery time* is used to specify facts such as "Once the service is unavailable it takes 24 hours to make it available again".

– *Availability for varying time regimes* is used to specify facts such as "Service availability is 99% from 9:00 to 12:00 and 60% at night and during week-ends", which is used for evaluating the Level of Uptime.
– *Percentage of conversations completed within defined performance level* is used to specify facts such as "A conversation is completed 99% times within 3 min., 0.15% within 1 hour, 0,05% within 3 hours", which is used for evaluating the Percentage of Timely Service Provisioning Requests.

2.2 Formalisation of metrics

A way for formalizing SLA metrics is to define *service metric functions* that aggregates a set of measurements taken for a service over all its allocated resources and with respect to a time window. These functions are either input to a monitoring platform that observes the deployment environment and reacts to changes or statically verified by means of techniques as those developed in Chapters 3 and 4. Let us discuss the two possibilities by means of two examples.

Let *service availablity* $\alpha(s,\tau)$, where s is the service name and τ is a time period, be

$$\alpha(s,\tau) = \frac{\text{actual service capacity}}{\text{contracted service capacity}}$$

Intuitively, $\alpha(s,\tau)$ gives the actual capability of s over a time period τ compared to the contracted SLA. The violation of a SLA is detected by periodically checking $\alpha(s,\tau)$ by means of a monitoring platform. For example, assume that the contracted SLA constrains a query service to complete 10 queries per second and take a monitoring window $\tau = 5$ minutes. This means that the expected contracted SLA is $10 \times 60 \times 5 = 3000$. Suppose we measure the service s during the monitoring window τ and we find a value of 2900. Then $\alpha(s,\tau) = \frac{2900}{3000} = 0.966$. If an acceptable violation (tolerance) range of 3% was negotiated with the customer, this means that s is under-capacity because $\alpha(s,\tau) < 1 - 0.03$. The definition of such service metrics is being investigated by FRH and will be used in the FRH case study (*c.f.* WP4) and reported in Deliverable **D2.3**.

The second example of a formal metric is defined in the context of the ATB model (See Section 2.3.1) and is the *best case delivery time of a data update to a mobile device*. If a user has an average bandwidth of B Mbit/s since a document D arrives into the cloud, the best case delivery time would be $t_{BestCase} = |D|/B$, assuming that the cloud processing time is 0. However, assuming that a user has given permission to the search application to consume up to 20% of the mobile device capacity, the best case time becomes $t_{BestCase} = |D|/(20\% * B)$. The average bandwidth is a measure that can be statically estimated by annotating the code with the size of data. Then, a behavioural interface can be extracted from the code with techniques similar to the one defined in Section 4, and by analysing such behavioural interfaces against the **Service data size metric** defined in Section 2.1 one can estimate the average (and the maximum) bandwidth needed for the user code.

2.3 Examples from the case studies

2.3.1 ATB model

The ATB model is an end-to-end model from cloud service to the mobile handset, developed in WP4. Figure 2.1 illustrates one of the most relevant metric used for evaluating the model: the *freshness of index data on the mobile device*. This metric provides time-related guarantees to mobile users on the interval between the moment in which a document has appeared in the cloud (C1 in Figure 2.1) to the moment in which it has been indexed and made available on the mobile device (CN in Figure 2.1).

The evaluation of the time from a data update (C1) to updated on-device results (CN) is related to some of the performance metrics of Section 2.1. In particular, this evaluation depends on the waiting times in the server (**Response Time**), the number of requests a server can receive (**Service Throughput**), and the time to transfer the data to the devices depending on the amount of data in the response and the bandwidth

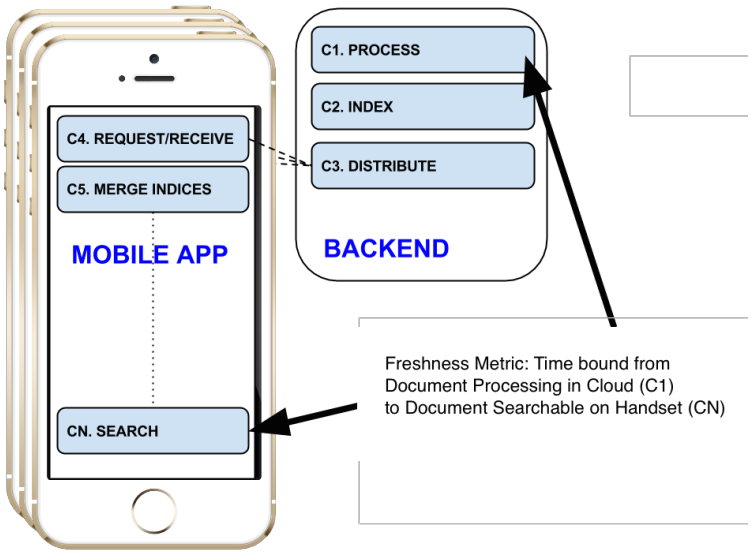

Figure 2.1: High-Level SLA

available (**Maximum Resource Capacity**). It is worth to remark that the evaluation of the time from a data update (C1) to updated on-device results (CN) must take into account the varying wireless bandwidth the user has on his mobile device (see the $t_{BestCase}$ metric defined in Section 2.2). Additionally, given that the cloud processing (C1) and indexing (C2) takes time, and the update might wait in a queue (either constrained by cloud or mobile device network capacity) before being distributed from the cloud towards mobile device (C3), there is likely to be additional and significant wait time since one will only send index updates with more than a single document. When the index arrives on the device (C4), there might be additional steps, *e.g.* merging index update (containing new document D with previous indices), before D is finally searchable (CN).

2.3.2 ENG model

One of the functionalities of the ENG case study developed in WP4 is the scheduler of the ETICS Resource Planning Module (RPM). This module has to schedule the execution of builds (*e.g.* of ant-based projects) on a set of virtual machines provided by a hybrid cloud infrastructure. In turn, each build project consists of a chain of complex tasks (e.g. javac, jar) that have to be scheduled in a predetermined given order. Each build is assigned a priority, deferral time, and window. The data type of a build project is the following

```
module Build;

data Project = NullProject | Project(
   String projectName, // parameters to configure the build
   List<Param> parameters, // list of targets defined by this project
   List<Target> targets, // default target to execute if no target is specified
   Target defaultTarget,
   List<Property> requirements,
   Int priority,
   Int queue,
   List<Param> resourceRequirements,
   Time deferralTime,
   Duration deferralWindow
);
```

To cope with large sets of builds, that contain thousands of tasks, resource requirements and dependencies, the RPM scheduler is designed as a concurrent system consisting of several schedulers each of which is given a set of builds to schedule.

Given a number of tasks n_T, and being D_s the amount of data required by each scheduler s, then the following requirement must be verified by RPM: $D_s \leq {n_T}^2$. That is, the amount of data required by each

RPM module must be kept below the square of the number of tasks being scheduled. The verification of this requirement will be undertaken by runtime monitoring systems and is related to capacity in Section 2.1.

2.3.3 FRH Model

The Fredhopper Cloud Services offer search and targeting facilities on a large product database to e-commerce companies (*c.f.* the FRH case study in WP4). These services are exposed at endpoints that are typically implemented to accept connections over HTTP. For example, one of the services offered by these endpoints is the Fredhopper Query API, which allows users to query over their product catalog via full text search[1] and faceted navigation[2].

A customer of FRH uses the Query API owns a *single* HTTP endpoint for searching the catalog and other operations. The Query API delivered to the customer is implemented by means of a number of resources (virtual machines) that are managed by a load balancer. In the present modeling, each resource is launched to serve *one* instance of Query API. That is, the sharing of resources among customers is disallowed.

When a customer signs an SLA contract with FRH, there is a clause in the contract that describes the Performance properties of the Query API. The usual metrics in the document are:

Service availability $\alpha(s, \tau)$ detailed in Section 2.2, measures the capacity and the availability of a service at FRH.

Query per Second (QPS) that defines the number of completed queries per second for a customer. An agreement is a bound on the expected QPS and forms the basis of many decisions (technical or legal) thereafter. The agreement is used by the operations team to set up an environment for the customer that includes the necessary resources described above. The agreement is additionally used by the support team to manage communications with the customer during the lifetime of the service for the customer. QPS is an instance of the **Service Throughput** of Section 2.1.

Query API proctime is the duration from when Query API receives a query request until the time when the result is completely written to the response or a failure is reported. This is an instance of the **Response Time**. The FRH `proctime` is determined based on the size of the data managed by the Fredhopper Query service. FRH always ensures a response from Query API. The following code defines the behavioural interface of Query API:

```
interface QueryService {
    @ ensures reply == True ; // the service always replies
    @ within const * length(data) ; // within a time proportional to the input data

    // Service definitions
}
```

[1] `http://en.wikipedia.org/wiki/Full_text_search`
[2] `http://en.wikipedia.org/wiki/Faceted_navigation`

Chapter 3

Behavioral Interfaces and Time

3.1 Overview

In this chapter, we address the formal verification of service contracts for virtualized services. We consider a simple setting with an interface language which specifies services, including their service contracts in the form of response time guarantees, and a simple object-oriented language, called μABS, for realizing these services. μABS is a restricted version of ABS to specify resource-aware virtualized services [7, 16, 15]. To support non-functional behavior, the language is based on a real-time semantics and associates deadlines with method calls. Virtualization is captured by associating execution capacities to dynamically created objects. Thus, the time required to execute a method activation depends not only on the actual parameters to the method call, but also on the execution capacity of the called object. This execution capacity reflects the processing power of virtual machine instances, which are created from within the service itself. We show in this chapter how to apply deductive verification techniques to ensure that *all local deadlines are met during the execution of a virtualized service.*

As an initial step, the work described in this chapter is restricted to sequential computation and synchronous method calls. We discuss at the end of the chapter about how to extend the work to a concurrent setting for ABS which is based on concurrent objects and asynchronous method calls.

3.2 Behavioral Interfaces with Response Time

To integrate service contracts and configuration parameters in service models, we aim at a *design by contract* methodology [17] for SLA-aware virtualized services. The methodology *incorporates SLA requirements in the interfaces at the application-level* to ensure the QoS expectations of clients. We consider an object-oriented setting with service-level interfaces given in a style akin to JML [10] and Fresco [20]: **requires**- and **ensures**-clauses express each method's functional pre- and postconditions. In addition, we further extend the interfaces with response time annotations by introducing a **within**-clause associated with the method, which summarises *response time guarantees.* The specification of methods in interfaces is illustrated in Figure 3.1.

Semantics

The semantics of μABS is stack-based, and is of the form: $stack \rightarrow stack'$ where a task stack is either $f \bullet q$ or **idle**, and both f and q are stack frames. The semantics is rather standard, and is given by a set of operational rules. We show here two of the most illustrative rules in Figure 3.2.

Objects are created with a given *capacity*, which expresses the processing cycles available to the object per time interval when executing its methods. The capacity of an object is the output of the evaluation of the expression e (cf. Rule R-New-Object). Time passes when a statement **job**(e) is executed on top of the task stack, where **job**(e) captures an execution requiring e processing cycles. A job abstracts from actual computations but may depend on state variables. The effect of executing this statement on an object

```
interface Service {
  @ requires φ(x);
  @ ensures ψ(return);
  @ within 5;
  Int method(Int x);
}
class Server () {
  @ requires x ≥ 0;
  @ within 4;
  Int method(Int x) { job(8); return x; }
}
// Main block:
{ Service y = new Server() with 2,
  Int z = y.method(5);
  print("Final result: "); println(z);
}
```

Figure 3.1: Behavioral interface with response time.

$$\frac{fresh(o) \quad e \Downarrow_\sigma = r \quad classOf(o) = C \quad capacity(o) = r}{\{\sigma | x = \textsf{new } C() \textsf{ with } e; sr\} \bullet q \rightarrow \{\sigma | x = o; sr\} \bullet q} \text{(R-New-Object)}$$

$$\frac{q' = \delta(\{\sigma | sr\} \bullet q, \frac{r'}{r}) \quad e \Downarrow_\sigma = r' \quad capacity(\textsf{this}) = r \quad r > 0}{\{\sigma | \textsf{job}(e); sr\} \bullet q \rightarrow q'} \text{(R-Job)}$$

Figure 3.2: Structural operational semantics

with capacity r, is that the local deadline of every task on the stack decreases by r'/r, where r' is the value resulting from evaluating e. The auxiliary function δ in Rule R-Job specifies how time advances in the system. It ensures the local deadline of all the subsequent tasks on the stack is updated.

A Proof System for Response Time Guarantees

Annotating non-functional properties in the interfaces, like response time on which we focus in this chapter, allows our approach to support the notion of design-by-contract compositionality for such properties. This together with Hoare-reasoning enables to *compositionally* verify virtualized services with respect to response time. We show the rule for reasoning about method definitions in μABS:

$$\frac{\{\phi \wedge deadline \geq e\} \ sr \ \{\psi \wedge deadline \geq 0\}}{\textsf{@requires } \phi; \ \textsf{@ensures } \psi; \ \textsf{@within } e; \ \mathsf{T}'' \ m \ (\overline{\mathsf{T}} \ \overline{x}) \ \{\overline{T'} \ \overline{x'}; sr\}} \text{(Method)}$$

It is is formalized using a Hoare triple $\{\phi\}$ sr $\{\psi\}$ with a standard partial correctness semantics: if the execution of sr starts in a state satisfying the precondition ϕ and the execution terminates, the result will be a state satisfying the postcondition ψ. sr is a sequence of statements followed by a **return**-statement.

The premise of Rule Method assumes that the execution of sr starts in a state where the **requires**-clause ϕ is satisfied and that the expected response time (*deadline*) is larger than expression e, where e is the specified response time guarantee from the **within**-clause. When the execution of sr terminates, the result will satisfy the **ensures**-clause ψ and the expected response time remains non-negative.

3.3 Conclusive Remarks

In this chapter, we discuss some initial ideas about applying program verification techniques to models of virtualized services. In particular, we focus on response time aspects of service contracts by summarising this type of non-functional properties of services in behavioral interfaces. This is formalized in μABS, which is a

simple object-oriented language. We develop a proof system for the deductive verification of timing properties to a setting of virtualized programs. The extension of service interfaces with response time guarantees, as described earlier in this chapter, allows a compositional design-by-contract approach to service contracts for virtualized systems. The details of behavioral interfaces with response time are further discussed in Appendix A.

As mentioned in the beginning of the chapter, the work is a preliminary step towards the development of formal verification of service contracts for virtualized services. A natural extension of the current work is to alleviate restrictions of sequential computations and synchronous method calls so as to support concurrency and asynchronous method calls. One complication of this extension is to calculate the response time of method calls, which in general includes the execution time of methods, and the waiting time for the execution in a concurrent setting. A worst-case cost analysis [2, 1] provides upper bounds on resource consumption of methods, including execution time as well as the length of a task queue, which provides an estimation of the maximum response time for methods. Furthermore, incorporating the worst-case cost analysis into the proof system allows replacing the job-statements used in the current work with code which reflects the actual computations. The extension to concurrency enables us to build concurrency models of ABS, which can be reasoned about with the automated deductive verification tool `KeY-ABS` that will be delivered as part of WP3.

Another interesting challenge, which remains to be investigated, is how to incorporate the global requirements which we find in many service-level agreements into a compositional proof system, such as the maximum number of end users.

Chapter 4

Behavioral Interfaces and Resources

4.1 Overview

In this chapter we propose a *static analysis technique* that computes upper bounds of virtual machine usages in a *concurrent* language with explicit creation and release operations of virtual machines. This language, which is consistent with `ABS`, features the delegation to other (ad-hoc or third party) code of the releasing of 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 behavioral types and return cost equations, and (*iii*) an automatic solver for the the cost equations.

Our technique may be also applied to estimate (heap) memory consumptions in `ABS` as well as other (object-oriented) programming languages.

4.2 Behavioural types with resource usages

The analysis of resource usage in a program is of great interest because an accurate assessment could reduce allocation costs and energy consumption. These two criteria are even more important today, in modern architectures like cloud computing or mobile devices, where resources, such as virtual machines, have hourly or monthly rates. In facts, 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 sample codes, extrapolating this information for fully real-life complex programs could be cumbersome and highly error-sensitive. While it is relatively easy to estimate worst-case costs for sample codes, extrapolating this information for fully real-life complex programs could be cumbersome and highly error-sensitive. The first attempts about the analysis of resource usages dates back to Wegbreit's pioneering work in 1975 [19], 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 [18, 14]).

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? Under the assumption that one can acquire a virtual machine that has been previously released. A solution to this problem 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 modelled 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:

- 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 the cost of methods like `double_release` in a precise way, we associate methods with abstract descriptions that also carry the information about the state of the parameters and their identities. These descriptions are called *behavioural types* and are formally connected to the programs by means of a typing system. For example, the behavioural type of `double_release` is

```
double_release α(β, γ) {
  β✓ [α ↦ ∅α, β ↦ ∅β, γ ↦ ∅γ]; // release β, if β is active
  γ✓ [α ↦ ∅α, β ↦ ∅⊥, γ ↦ ∅γ]; // release γ, if γ is active
} −, {β, γ}
```

where types $\beta^{\checkmark}$ and $\gamma^{\checkmark}$ carry an environment recording state of the names and their identity.

We therefore analyse behavioural type descriptions by translating them in codes that are adequate for powerful off-the-shelf solvers that are developed at TUD – the `CoFloCo` tool [11] – or at UCM – the `PUBS` tool [3]. As discussed in [8], 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 the two arguments equal then its cost cannot be computed by a function taking two arguments, but it must be computed by a function with only one argument. 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. For example, in case of `double_release`, the output of our translation is

$$
\begin{array}{ll}
double\_release_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\
double\_release_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 \neq \bot] \\
double\_release_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
double\_release_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 \neq \bot] \\
 & \\
double\_release_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\
double\_release_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = double\_release_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) & [\alpha_1 = \partial] \\
double\_release_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 = \top]
\end{array}
$$

where the function `CREL` is defined as follows:

$$\texttt{CREL}(\alpha) \;=\; \begin{cases} \texttt{-1} & \alpha = \top \\ \texttt{0} & \text{otherwise} \end{cases}$$

It is worth to observe that we use the metaphor of cloud computing and virtual machines. Actually our technique addresses every type of resources that retain operations of acquire (or creation) and release, such as heap usages in concurrent object-oriented languages.

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, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors, *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Static Analysis*, pages 221–237. Springer-Verlag, 2008.

[4] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

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

[6] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, Lecture Notes in Computer Science. Springer, 2015. To appear.

[7] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter 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.

[8] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *Static Analysis*, pages 405–421. Springer, 2012.

[9] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 140–155. Springer, 2014.

[10] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[11] 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, Singapore*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, November 2014.

[12] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling*, 2014. To Appear.

[13] 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`.

[14] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14, 2012.

[15] Einar Broch Johnsen, Rudolf Schlatte, and S.L̃izeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In Toshiaki Aoki and Kenji Tagushi, editors, *Proc. 14th International Conference on Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, November 2012.

[16] Einar Broch Johnsen, Rudolf Schlatte, and S.L̃izeth 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.

[17] Bertrand Meyer. Design by contract: The Eiffel method. In *TOOLS (26)*, page 446, 1998.

[18] Huu Hai Nguyen, Wei ngan Chin, Shengchao Qin, and Martin Rinard. Memory usage verification for oo programs. In *In SAS 05*, pages 70–86. Springer, 2005.

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

[20] Alan Wills. Capsules and types in Fresco: Program verification in Smalltalk. In Pierre America, editor, *European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, 1991.

Glossary

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

μABS A subset of ABS, which is an object-oriented sequential lanugage with synchronous method calls. It formalises service level agreements and service contracts.

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.

Cost of a program The amount of resources necessary for the (proper) execution of a program.

Elasticity Is the ability of a software component of dynamically acquiring and releasing resources, aiming to minimize the difference between the resources reserved and the resources actually in use.

Full-fledged release The capacity of any software component to release a known resource independently of whether this resource was acquired locally or by another part.

Peak Cost Is the (worst case) number of resources acquired at the same time during an execution.

PUBS Practical Upper Bounds Solver. A tool that automatically calculates (an upper bound of) the cost of a program from a given set of equations describing the costs of its components.

Net Cost Is the (worst case) number of locally acquired resources that remain unreleased after the execution.

Appendix A

Meeting Deadlines, Elastically

Meeting Deadlines, Elastically *

Einar Broch Johnsen, Ka I Pun, Martin Steffen,
S. Lizeth Tapia Tarifa, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, Norway
{einarj,violet,msteffen,sltarifa,ingridcy}@ifi.uio.no

1 Introduction

A cloud consists of virtual computers that are accessed remotely for data storage and processing. The cloud is emerging as an economically interesting model for enterprises of all sizes, due to an undeniable added value and compelling business drivers [11]. One such driver is *elasticity*: businesses pay for computing resources when needed, instead of provisioning in advance with huge upfront investments. New resources such as processing power or memory can be added to a virtual computer on the fly, or an additional virtual computer can be provided to the client application. Going beyond shared storage, the main potential in cloud computing lies in its scalable virtualized framework for data processing. If a service uses cloud-based processing, its capacity can be automatically adjusted when new users arrive. Another driver is *agility*: new services can be deployed on the market quickly and flexibly at limited cost. This allows a service to handle its users in a flexible manner without requiring initial investments in hardware before the service can be launched.

Today, software is often designed while completely ignoring deployment or based on very specific assumptions, e.g., the size of data structures, the amount of random access memory, and the number of processors. For the software developer, cloud computing brings new challenges and opportunities [21]:

- **Empowering the Designer.** The elasticity of software executed in the cloud gives designers far reaching control over the execution environment's resource parameters, e.g., the number and kind of processors, the amount of memory and storage capacity, and the bandwidth. In principle, these parameters can even be adjusted at runtime. The owner of a cloud service can not only deploy and run software, but also control trade-offs between the incurred cost and the delivered quality-of-service.
- **Deployment Aspects at Design Time.** The impact of cloud computing on software design goes beyond scalability. Deployment decisions are traditionally made at the end of a software development process: the developers first design the functionality of a service, then the required resources are determined, and finally a service level agreement regulates the provisioning of these resources. In cloud computing, this can have severe consequences:

* This work was done in the context of the EU project FP7-610582 *ENVISAGE: Engineering Virtualized Services* (http://www.envisage-project.eu)

a program which does not scale usually requires extensive design changes when scalability was not considered a priori.

To realize cloud computing's potential, software must be *designed for scalability*. This leads to a new *software engineering challenge*: how can the validation of deployment decisions be pushed up to the modeling phase of the software development chain without convoluting the design with deployment details?

The EU project Envisage addresses this challenge by extending a design by contract approach to service-level agreements for resource-aware virtualized services. The functionality is represented in a *client layer*. A *provisioning layer* makes resources available to the client layer and determines how much memory, processing power, and bandwidth can be used. A *service level agreement* (SLA) is a legal document that clarifies what resources the provisioning layer should make available to the client service, what they will cost, and the penalties for breach of agreement. A typical SLA covers two different aspects: (i) the mutual legal obligations and consequences in case of a breach of contract, which we call the *legal contract*; (ii) the technical parameters and cost figures of the offered services, which we call the *service contract*.

This paper discusses some initial ideas about applying program verification techniques to models of virtualized services. We consider response time aspects of service contracts and extend JML-like interfaces with response time annotations. This is formalized using μABS; μABS is a restricted version of ABS [25], an executable object-oriented modeling language used in the Envisage project to specify resource-aware virtualized services [4, 26, 27]. In particular, the work discussed in this paper is restricted to sequential computation and synchronous method calls whereas ABS is based on concurrent objects and asynchronous method calls. In future work, we hope to alleviate these restrictions.

Paper organization. Section 2 introduces service interfaces with response-time annotations; Sect. 3 introduces the syntax of μABS, the modeling language considered in this paper; Sect. 4 demonstrates the approach on an example; Sect. 5 develops a Hoare-style proof system for μABS; Sect. 6 discusses related work; and Sect. 7 concludes the paper.

2 Service-Level Interfaces

Service level agreements express non-functional properties of services (service contracts), and their associated penalties (legal contracts). Examples are high water marks (e.g., number of users), system availability, and service response time. Our focus is on service contract aspects of client-level SLAs, and on how these can be integrated in models of virtualized services. Such an integration allows a formal understanding of service contracts and of their relationship to the performance metrics and configuration parameters of the deployed services. Today, client-level SLAs do not allow the potential resource usage of a service to be determined or adapted when unforeseen changes to resources occur. This is because user-level SLAs are not explicitly related to actual performance metrics and configuration parameters of the services. The integration of service

```
type Photo = Rat; // size of the file

interface PhotoService {
    @requires ∀ p:Photo · p ∈ film && p < 4000;
    @ensures reply == True;
    @within 4∗length(film) + 10;
    Bool request(List<Photo> film);
}
```

Fig. 1. A photo printing shop in μABS.

contracts and configuration parameters in service models enables the design of resource-aware services which embody application-specific resource management strategies [21].

The term *design by contract* was coined by Bertrand Meyer referring to the contractual obligations that arise when objects invoke methods [33]: only if a caller can ensure that certain behavioral conditions hold before the method is activated (the precondition), it is ensured that the method results in a specified state when it completes (the postcondition). Design by contract enables software to be organized as encapsulated services with interfaces specifying the contract between the service and its clients. Clients can "program to interfaces"; they can use a service without knowing its implementation. We aim at a design by contract methodology for SLA-aware virtualized services, which *incorporates SLA requirements in the interfaces at the application-level* to ensure the QoS expectations of clients.

We consider an object-oriented setting with service-level interfaces given in a style akin to JML [10] and Fresco [46]; **requires**- and **ensures**-clauses express each method's functional pre- and postconditions. In addition, a *response time guarantee* is expressed in a **within**-clause associated with the method. The specification of methods in interfaces is illustrated in Figure 1.

3 A Kernel Language for Virtualized Computing

The μABS language supports modeling the deployment of objects on virtual machines with different processing capacities, simplifying ABS [4,25,27]: conceptually, each object in μABS has a dedicated processor with a given processing capacity. In contrast to ABS, execution in μABS is sequential and the communication between named objects is synchronous, which means that a method call blocks the caller until the callee finishes its execution. Objects are dynamically created instances of classes, and share a common thread of execution where at most one task is *active* and the others are waiting to be executed on the task stack. μABS is strongly typed: for well-typed programs, invoked methods are understood by the called object. μABS includes the types Capacity, Cost, and Duration which all extend Rat with an element infinite: Capacity captures the processing capacity of virtual machines per time interval, Cost the processing cost of executions, and Duration time intervals.

Syntactic categories	*Definitions*
C, I, m in Names	$P ::= \overline{IF}\ \overline{CL}\ \{\overline{T\ x};\ sr\}$
s in Statement	$T ::= C \mid I \mid$ Capacity $\mid$ Cost $\mid$ Duration $\mid$ Bool $\mid$ Rat
x in Variables	$IF ::= \textbf{interface}\ I\ \{\ \overline{Sg}\ \}$
k in Capacity	$Sg ::= \overline{Spec}\ T\ m\ (\overline{T}\ \overline{x})$
c in Cost	$Spec ::= \textbf{@requires}\ \phi; \mid \textbf{@ensures}\ \phi; \mid \textbf{@within}\ \phi;$
d in Duration	$CL ::= \textbf{class}\ C\ (\overline{T}\ \overline{x})\ \{\ \overline{M}\}$
b in Bool	$M ::= Sg\ \{\ \overline{T}\ \overline{x};\ sr\ \}$
i in Rat	$sr ::= s; \textbf{return}\ e \mid \textbf{return}\ e$
	$s ::= s; s \mid x = rhs \mid \textbf{job}(e) \mid \textbf{if}\ e\ \{s\}\ \textbf{else}\ \{s\}$
	$rhs ::= e \mid \textbf{new}\ C(\overline{e})\ \textbf{with}\ e \mid e.m(\overline{x})$
	$e ::= \textbf{this} \mid \textbf{capacity} \mid \textbf{deadline} \mid x \mid v \mid e\ op\ e$

Fig. 2. μABS syntax for the object level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories.

Figure 2 presents the syntax of μABS. A *program* P consists of interface and class definitions, and a main block $\{\overline{T\ x};\ sr\}$. Interfaces IF have a name I and method signatures $\overline{Sg}$. Classes CL have a name C, optional formal parameters $\overline{T}\ \overline{x}$, and methods $\overline{M}$. A method signature Sg has a list of specifications $\overline{Spec}$, a return type T, a method name m, and formal parameters $\overline{x}$ of types $\overline{T}$. In specifications (see Sect. 2), assertions ϕ express properties of local variables in an assertion language extending the expressions e with logical variables and operators in a standard way; a reserved variable *reply* captures the method's return value. A method M has a signature Sg, a list of local variable declarations $\overline{x}$ of types $\overline{T}$, and statements sr. Statements may access local variables and the formal parameters of the class and the method.

Statements are standard, except **job**(e) which captures an execution requiring e processing cycles. A job abstracts from actual computations but may depend on state variables. *Right-hand sides rhs* include expressions e, object creation **new** $C\ (\overline{e})$ **with** e and synchronous method calls $e.m(\overline{x})$. Objects are created with a given *capacity*, which expresses the processing cycles available to the object per time interval when executing its methods. Method calls in μABS are *blocking*. Expressions e include operations over declared variables x and values v. Among values, b has type Bool, i has type Rat (e.g., $5/7$), k has type Capacity, c has type Cost, and d has type Duration. Among binary operators op on expressions, note that division c/k has type Duration. Expressions also includes the following reserved read-only variables: **this** refers to the object identifier, **capacity** refers to the processing speed (amount of resources per time interval) of the object, and **deadline** refers to the local deadline of the current method. (We assume that all programs are well-typed and include further functional expressions and data types when needed in the example.)

Time. μABS has a dense time model, captured by the type Duration. The language is not based on a (global) clock, instead each method activation has an associated local counter **deadline**, which decreases when time passes. Time passes when a statement **job**(e) is executed on top of the task stack. The effect of executing this statement on an object with capacity k, is that the local deadline of every task on the stack decreases by c/k, where c is the value resulting from evaluating e. The initial value of the **deadline** counter stems from the service

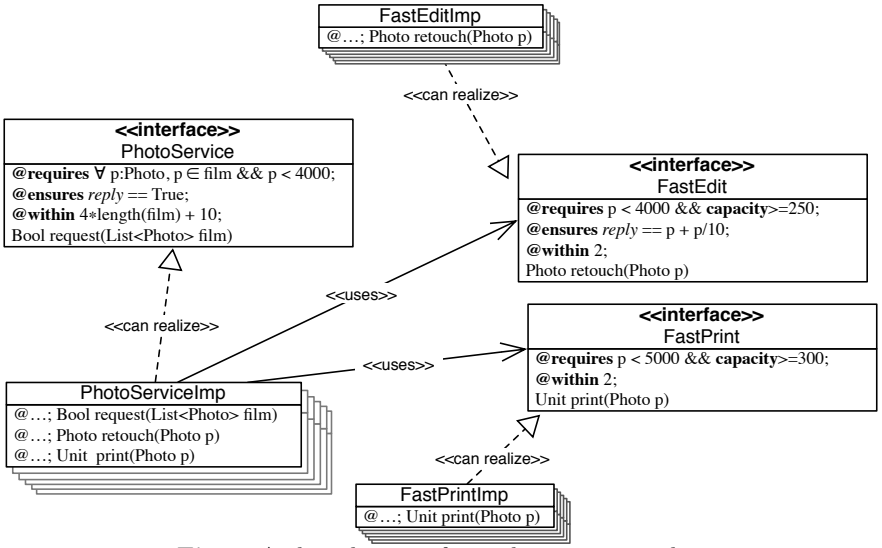

Fig. 3. A class diagram for a photo printing shop

contract; thus, a local counter which becomes negative represents a breach of the local service contract. For brevity, we omit the formal semantics.

4 Example: A Photo Printing Shop

Let us consider a *photo shop* service which *retouches* and *prints* photos. It is cheaper for the photo shop service to retouch and print photos locally, but it can only deal with low resolution photos in time. For larger photos, the photo shop service relies on using a faster and more expensive laboratory in order to guarantee that all processing deadlines are met successfully.

In this example, a film is represented as a list of photos and, for simplicity, a photo by the size of the corresponding file. As shown in the class diagram of Figure 3, an interface PhotoService provides a single method request which handles customer requests to the photo shop service. The interface is implemented by a class PhotoServiceImp, which has methods retouch for retouching and print for printing a photo, in addition to the request method of the interface. For faster processing, two interfaces FastEdit and FastPrint, which also provide the methods retouch and print, may be used by PhotoServiceImp. The sequence diagram in Figure 4 shows how a photo is first *retouched*, then *printed*. The tasks of retouching and printing are done locally if possible, otherwise they are forwarded to and executed by objects with higher capacities.

The μABS model of the example (Figure 5) follows the design by contract approach and provides a contract for every method declaration in an interface and method definition in a class. These specifications are intended to guarantee

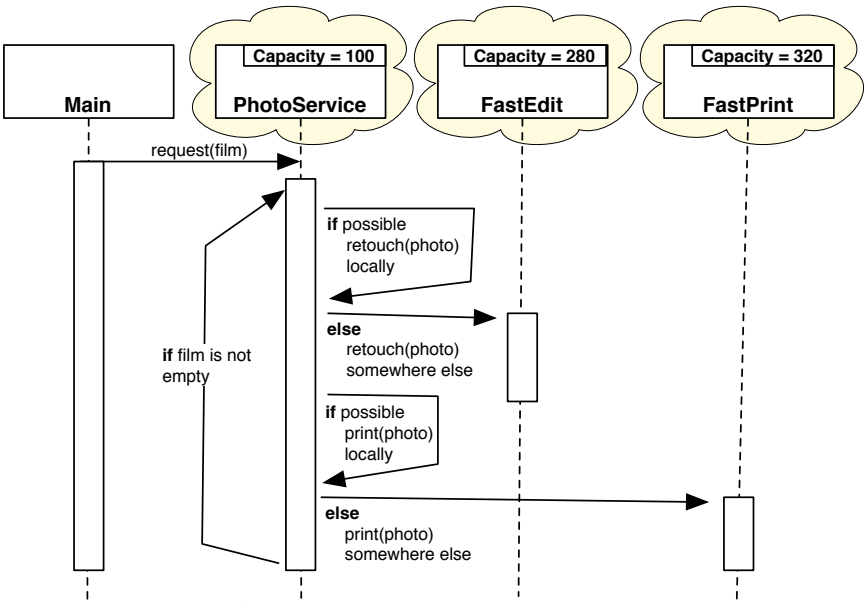

Fig. 4. A sequence diagram for a photo printing shop

that a request to a PhotoService object will not break the specified contract. Looking closer at the contract for request, we see that the response time of a request(film) call depends on the length of the film and assumes that the size of every photo contained in the film is smaller than 4000. The implementation of the request method is as follows: Take the first photo in the film (by applying the function head(film)) and check if this photo is low resolution compared to the capacity of the PhotoService object, represented by a size smaller than 500 and a capacity of at least 100, respectively. In this case, the retouch can be done locally, otherwise retouch is done by an auxiliary FastEdit object. A similar procedure applies to printing the retouched photos. Thus, photos of small sizes are retouched and printed locally, while photos with bigger sizes are sent to be retouched and printed externally. The implementations of the different methods are abstractly captured using **job** statements.

5 Proof System

The proof system for μABS is formalized as Hoare triples [5, 22] $\{\phi\}\ s\ \{\psi\}$ with a standard partial correctness semantics: if the execution of s starts in a state satisfying the precondition ϕ and the execution terminates, the result will be a state satisfying the postcondition ψ. In this paper, we are particularly interested in assertions about the *deadline* variables of method activations.

The reasoning rules for μABS are presented in Figure 6. Reasoning about sequential composition, conditional, and assignment is standard, and captured

```
type Photo = Rat; // size of the file

interface FastEdit {
    @requires p < 4000 && capacity>=250; @ensures reply == p + p/10; @within 2;
    Photo retouch(Photo p);}
class FastEditImp {
    @requires p < 4000 && capacity>=200; @ensures reply == p + p/10; @within 2;
    Photo retouch(Photo p) {job(200); return (p + p/10)}}

interface FastPrint {
    @requires p < 5000 && capacity>=300; @within 2;
    Unit print(Photo p);}
class FastPrintImp {
    @requires p < 5000 && capacity>=250; @within 2
    Unit print(Photo p) {job(250);return unit}}

interface PhotoService {
    @requires ∀ p:Photo, p ∈ film && p < 4000;
    @ensures reply == True; @within 4∗length(film) + 10;
    Bool request(List<Photo> film);}
class PhotoServiceImp(FastEdit edit,FastPrint print) {
    @requires ∀ p:Photo, p ∈ film && p < 4000;
    @ensures reply == True; @within 4∗length(film)+1;
    Bool request(List<Photo> film) {
        Photo p = 0;
        if (film != Nil){
            p = head(film);
            if (p < 500 && capacity>=100){ p = this.retouch(p);}
            else{p = edit.retouch(p);}
            if ( p < 600 && capacity>=100){this.print(p);}
            else{print.print(p);}
            this.request(tail(film));}
        else{ job(1);}
        return (deadline >= 0) }

    @requires p < 500 && capacity>=100; @ensures reply == p + p/20; @within 1;
    Photo retouch(Photo p) {job(100); return (p + p/20)}

    @requires p < 600 && capacity()>=100; @within 1;
    Unit print(Photo p) { job(100); return unit}}
```

Fig. 5. A photo printing shop in μABS

by the rules Comp, Cond, and Assign, respectively. Time passes when **job**(e) is executed; **job**(e) has a duration e/cap on an object with capacity cap. The assertion in Rule Job ensures that this duration is included in the response time after executing **job**(e). The subsumption rule allows to strengthen the precondition and weaken the postcondition. For method definitions, the premise of Rule Method assumes that the execution of sr starts in a state where the **requires**-clause ϕ is satisfied and that the expected response time ($deadline$) is larger than expression e, where e is the specified response time guarantee from the **within**-clause. When the execution of sr terminates, the result will satisfy the **ensures**-clause ψ and the expected response time remains non-negative. For method invocations in Rule Call, the specification of the method is updated

$$\frac{\{\phi \wedge \mathit{deadline} \geq e\}\ sr\ \{\psi \wedge \mathit{deadline} \geq 0\}}{\begin{array}{c}\textbf{@requires}\ \phi;\ \textbf{@ensures}\ \psi;\ \textbf{@within}\ e;\\ \mathsf{T}''\ m\ (\overline{\mathsf{T}}\ \overline{x})\ \{\overline{T'}\ \overline{x'}; sr\}\end{array}}\ (\textsc{Method}) \qquad \frac{\{\phi\}\ s; \mathit{reply} = e\ \{\psi\}}{\{\phi\}\ s; \textbf{return}\ e\ \{\psi\}}\ (\textsc{Return})$$

$$\frac{\{\phi\} s_1 \{\psi'\} \quad \{\psi'\} s_2 \{\psi\}}{\{\phi\} s_1; s_2 \{\psi\}}\ (\textsc{Comp}) \qquad \frac{\{\phi \wedge b\} s_1 \{\psi\} \quad \{\phi \wedge \neg b\} s_2 \{\psi\}}{\{\phi\}\ \textbf{if}\ b\ \{s_1\}\ \textbf{else}\ \{s_2\}\ \{\psi\}}\ (\textsc{Cond}) \qquad \frac{\{\phi'\}\ s\ \{\psi'\} \quad \phi' \Rightarrow \phi \quad \psi' \Rightarrow \psi}{\{\phi\}\ s\ \{\psi\}}\ (\textsc{Subsumption})$$

$$\{\phi[x \mapsto e]\}\ x = e\ \{\phi\}\ (\textsc{Assign}) \qquad \{\phi[\mathit{deadline} \mapsto \mathit{deadline} - (e/\mathrm{cap})]\}\ \textbf{job}(e)\ \{\phi\}\ (\textsc{Job})$$

$$\frac{\begin{array}{c}\mathit{fresh}(\alpha)\\ \phi' = \phi[x \mapsto \alpha]\\ T = \mathit{typeOf}(x)\\ \phi' \Rightarrow \mathit{implements}(C, T, e)\end{array}}{\{\phi'\}\ x = \textbf{new}\ C(\overline{e})\ \textbf{with}\ e\ \{\phi\}}\ (\textsc{New}) \qquad \frac{\begin{array}{c}\mathit{fresh}(\alpha, \beta) \quad T = \mathit{typeOf}(e)\\ \phi' = \phi[x \mapsto \alpha, \mathit{deadline} \mapsto \mathit{deadline} - \beta]\\ \phi' \Rightarrow \mathit{requires}(T, m)[\overline{fp} \mapsto \overline{e}]\\ \phi_1 = \mathit{ensures}(T, m)[\overline{fp} \mapsto \overline{e}, \mathit{reply} \mapsto \alpha]\\ \phi_2 = \mathit{within}(T, m)[\overline{fp} \mapsto \overline{e}, \mathit{deadline} \mapsto \beta]\end{array}}{\{\phi' \wedge \phi_1 \wedge \phi_2\}\ x = e.m(\overline{e})\ \{\phi\}}\ (\textsc{Call})$$

Fig. 6. Proof system for μABS

by substituting the formal parameters $\overline{fp}$ by the input expressions $\overline{e}$. The logical variables for the return value of the method (*reply*) and of the expected response time are renamed with fresh variables α and β, respectively. To avoid name clashes between scopes, we assume renaming of of other variables as necessary. Object creation (in Rule New) is handled similarly to assignment. The precondition ensures that the newly created object of a class C with capacity e correctly implements interface T, where T is the type of x. (Note that the class instance may or may not implement an interface, depending on its capacity.) If a method has a return value, expression e in the return statement will be assigned to the logical variable *reply* in Rule Return, and can be handled by the standard assignment axiom in Rule Assign.

We show in Equation 3 the skeleton of the proof for the method `request` in Figure 5 by using the proof system presented in Figure 6. Let *sr* refers to the method body of `request` and *s* is *sr* without the return statement. In addition,

$$\begin{array}{c}\psi = \mathit{reply} == \texttt{True}, \quad \psi_1 = \psi \wedge \mathit{deadline} \geq 0,\\ \phi = \forall p : \mathrm{Photo},\ p \in \mathit{film}\ \wedge\ p < 4000, \quad \text{and} \quad e = 4 * \texttt{length}(\mathit{film}) + 10\end{array} \tag{1}$$

We further assume that

$$\psi_2 = \mathit{reply} == \mathit{deadline} \geq 0 \wedge \mathit{deadline} \geq 0 \tag{2}$$

be the postcondition of the assignment $\mathit{reply} = \mathit{deadline} \geq 0$.

By Rule METHOD, the assertions ϕ and $deadline > e$ serve as the precondition of the whole method body sr, where ϕ and e are defined in the **requires**- and **within**-clauses in the definition of the method `request` in Figure 5. The postcondition of the method body consists of ψ, which is specified in **ensures**-clause as $reply == \texttt{True}$, and the expression $deadline \geq 0$. Rule RETURN converts the **return** statement into a statement where the expression $deadline \geq 0$ is assigned to the logical variable *reply*. Then, by the assignment axiom ASSIGN, and with the postcondition ψ_2 assumed in Equation 2, the precondition ψ_3 is the postcondition with the logical variable *reply* substituted with the expression $deadline \geq 0$, and thus $\psi_3 = \texttt{True} \wedge deadline \geq 0$. By using Rule SUBSUMPTION, the postcondition ψ_2 is weakened to the given postcondition ψ_1. By Rule COMP, the assertion ψ_3 is also the postcondition of the statement s.

$$\dfrac{\dfrac{\vdots}{\{\phi \wedge deadline > e\}\ s\ \{\psi_3\}} \qquad \dfrac{\{\psi_3\}\ reply = deadline \geq 0\ \{\psi_2\} \quad \psi_2 \Rightarrow \psi_1}{\{\psi_3\}\ reply = deadline \geq 0\ \{\psi_1\}}}{\dfrac{\{\phi \wedge deadline > e\}\ s; reply = deadline \geq 0\ \{\psi_1\}}{\{\phi \wedge deadline > e\}\ s; \textbf{return}(deadline \geq 0)\ \{\psi_1\}}} \tag{3}$$

@requires ϕ; **@ensures** ψ; **@within** e;
Bool `request`(`List`⟨`Photo`⟩ `film`){sr}

For brevity, the rest of the proof is omitted in the paper, which can be completed by repeatedly applying the corresponding rules from the above proof system.

6 Related Work

The work presented in this paper is related to the ABS modeling language and its extension to virtualized computing on the cloud in the Envisage project. The ABS [25] language and its extensions with time [9], deployment component and resource-awareness [27] provide a formal basis for modeling virtualized computing. ABS has been used in two larger case studies addressing resource management in the cloud by combining simulation techniques and cost analysis, but not by means of deductive verification techniques; a model of the Montage case study [13] is presented in [26] and compared to results from specialized simulation tools and a large ABS model of the Fredhopper Replication Server has been calibrated using COSTABS [3] (a cost analysis tool for ABS) and compared to measurements on the deployed system in [4,12]. Related techniques for modeling deployment may be found in an extension of VDM++ for embedded real-time systems [45]. In this extension, static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain. Whereas ABS has been designed to support compositional verification based on traces [14], neither ABS nor VDM++ supports deductive verification of non-functional properties today.

Assertional proof systems addressing timed properties, and in particular upper bounds on execution times of systems, have been developed, the earliest example perhaps being [41]. Another early example to reason about real-time is Nielson's extension of classical Hoare-style reasoning to verify timed properties of a given program's execution [36, 37]. Soundness and (relative) completeness for of the proof rules of a simple while-language are shown. Shaw [40] presents Hoare logic rules to reason about the passage of time, in particular to obtain upper and lower bounds on the execution times of sequential, but also of concurrent programs.

Hooman employs assertional reasoning and Hoare logic [23] to reason about concurrent programs, covering different communication and synchronization patterns, including shared-variable concurrency and message passing using asynchronous channels. The logic introduces a dense time domain (i.e., the non-negative reals, including ∞) and assumes conceptually, for the purpose of reasoning, a single, global clock. The language for which the proof system is developed, is a small calculus, focussing on time and concurrency, where a **delay**-statement can be used to let time pass. This is comparable to the **job**-expression in our paper, but directly associates a duration with the job. In contrast, we associate a cost with the job, and the duration depends on the execution capacity of the deployed object. Timed reasoning using Dijkstra's weakest-precondition formulation of Hoare logic can be found in [19]. Another classical assertional formalism, Lamport's *temporal logic of actions* TLA [1,32], has likewise been extended with the ability to reason about time [31]. Similar to the presentation here, the logical systems are generally given by a set of derivation rules, given in a classical pre-/post-condition style. Thus, the approaches, in the style of Hoare-reasoning, are compositional in that timing information for composed programs, including procedure calls, is derived from that of more basic statements. While being structural in allowing syntax-directed reasoning, these formalisms do not explore a notion of timed interfaces as part of the programming calculus. Thus they do not support the notion of design-by-contract compositionality for non-functional properties that has been suggested in this paper.

Besides the theoretical development of proof systems for real-time properties, corresponding reasoning support has also been implemented within theorem provers and proof-assistants, for instance for PVS in [15] (using the duration calculus), and HOL [18]. An interesting approach for *compositional* reasoning about timed system is developed in [16]. As its logical foundation, the methodology uses TRIO [17], a general-purpose specification language based on first-order linear temporal logic. In addition, TRIO supports object-oriented structuring mechanisms such as classes and interfaces, inheritance, and encapsulation. To reason about open systems, i.e., to support modular or compositional reasoning, the methodology is based on a rely/guarantee formalization and corresponding proof rules are implemented within PVS. Similarly, a rely/guarantee approach for compositional verification in linear-time temporal logics is developed in [28, 44]. A further compositional approach for the verification of real-time systems is reported in [24], but without making use of a rely/guarantee framework.

Refinement-based frameworks are another successful design methodology for complex system, orthogonal to compositional approaches. Aiming at a correct-by-construction methodology, their formal underpinning often rests on various refinement calculi [6, 34, 35]. Refinement-based frameworks have also been developed for timed systems. In particular, Kaisa Sere and her co-authors [8] extended the well-known formal modeling, verification, and refinement framework Event-B [2] with a notion of time, resulting in a formal transformational design approach where the proof-obligations resulting from the timing part in the refinement steps are captured by timed automata and verified by the Uppaal tool [7].

The Java modeling language JML [10] is a well-known interface specification language for Java which was used as the basis for the interface specification of service contracts in our paper. Extensions of JML have been proposed to capture timed properties and to support component-based reasoning about temporal properties [29, 30]. These extensions have been used to modularly verify so-called performance correctness [42, 43]). For this purpose, JML's interface specification language is extended with a special **duration**-clause, to express timing constraints. The JML-based treatment of time is abstract insofar as it formalizes the temporal behavior of programs in terms of abstract "JVM cycles". Targeting specifically safety critical systems programmed in SCJ (Safety-critical Java), SafeJML [20] re-interprets the **duration**-clause to mean the worst-case execution time of methods concretely in terms of absolute time units. For a specific hardware implementation for the JVM for real-time applications, [39] presents a different WCET analysis [38] for Java. The approach does not use full-fledged logical reasoning or theorem proving, but is a static analysis based on integer linear programming and works at the byte-code level. We are not aware of work relating real-time proof systems to virtualized software, as addressed in this paper.

7 Concluding Remarks

Cloud computing provides an elastic but metered execution environment for virtualized services. Services pay for the resources they lease on the cloud, and new resources can be elastically added as required to offer the service to a varying number of end users at an appropriate service quality. In order to make use of the elasticity of the cloud, the services need to be *scalable*. A service which does not scale well may require a complete redesign of its business code. A *virtualized* service is able to adapt to the elasticity provided by the cloud. We believe that the deployment strategy of virtualized services and the assessment of their scalability should form an integral part of the service design phase, and not be assessed a posteriori after the development of the business code as it is done today. The design of virtualized services provides new challenges for software engineering and formal methods.

Virtualization empowers the designer by providing far-reaching control over the resource parameters of the execution environment. By incorporating a re-

source management strategy which fully exploits the elasticity of the cloud into the service, *resource-aware* virtualized services are able to balance the service contracts that they offer to their end users, to the metered cost of deploying the services. For resource-aware virtualized services, the integration of resource management policies in the design of the service at an early development stage seems even more important.

In this paper, we pursue a line of research addressing the formal verification of service contracts for virtualized services. We have considered a very simple setting with an interface language which specifies services, including their service contracts in the form of response time guarantees, and a simple object-oriented language for realizing these services. To support non-functional behavior, the language is based on a real-time semantics and associates deadlines with method calls. Virtualization is captured by the fact that objects are dynamically created with associated execution capacities. Thus, the time required to execute a method activation depends not only on the actual parameters to the method call, but also on the execution capacity of the called object. This execution capacity reflects the processing power of virtual machine instances, which are created from within the service itself. The objective of the proof system proposed in this paper is to apply deductive verification techniques to ensure that *all local deadlines are met during the execution of a virtualized service*. This proof system builds on previous work for real-time systems, and recasts the deductive verification of timing properties to a setting of virtualized programs. The extension of service interfaces with response-time guarantees, as proposed in this paper, allows a compositional design-by-contract approach to service contracts for virtualized systems.

Several challenges to the proposed approach are left for future work, in particular the extension to concurrency and asynchronous method calls, but also the incorporation of code which reflects the actual computations (replacing the job-statements of this paper). In this case, the abstraction to job-statements could be done by incorporating a worst-case cost analysis [3] into the proof system. Another interesting challenge, which remains to be investigated, is how to incorporate the global requirements which we find in many service-level agreements into a compositional proof system, such as the maximum number of end users.

References

1. Martín Abadi and Leslie Lamport. An old-fashioned recipe for real-time. In Jaco W. de Bakker, Cees Huizing, and Willem-Paul de Roever, editors, *Real-Time: Theory in Practice (REX Workshop)*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 1991.
2. Jean-Raymond Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
3. Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors, *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.

4. Elvira Albert, Frank S. de Boer, Reinar Hähnle, Einar Broch Johnsen, Rudolf Schlatte, S. Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. *J. of Service-Oriented Computing and Applications*, 2013. Springer Online First, DOI 10.1007/s11761-013-0148-0.
5. Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer, 3rd edition, 2009.
6. Ralph-Johan R. Back. *On the Correctness of Refinement in Program Development*. PhD thesis, Department of computer Science, University of Helsinki, 1978.
7. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang Yi. UPPAAL — a tool-suite for the automatic verification of real-time systems. In R. Alur, T. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1996.
8. Jesper Berthing, Pontus Boström, Kaisa Sere, Leonidas Tsiopoulos, and Jüri Vain. Refinement-based development of timed systems. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods – 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, volume 7321 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2012.
9. Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
10. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
11. Rajkumar Buyya, Chee S. Yeo, Srikumar Venugopal, James Broberg, and Ivona 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.
12. Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. In *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC)*, volume 7592 of *Lecture Notes in Computer Science*, pages 91–106. Springer, September 2012.
13. Ewa Deelman, Gurmeet Singh, Miron Livny, G. Bruce Berriman, and John Good. The cost of doing science on the cloud: The Montage example. In *Proceedings of the Conference on High Performance Computing (SC'08)*, pages 1–12. IEEE/ACM, 2008.
14. Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
15. Simon Fowler and Andy Wellings. Formal analysis of a real-time kernel specification. In Bengt Jonsson and Joachim Parrow, editors, *Proceedings of FTRTFT'96*, volume 1135 of *Lecture Notes in Computer Science*, pages 440–458. Springer, 1996.
16. Carlo A. Furia, Matteo Rossi, Dino Mandrioli, and Angelo Morzenti. Automated compositional proofs for real-time systems. *Theoretical Computer Science*, 376(3):164–184, 2007.
17. Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. TRIO, a logic language for executable specifications of realtime systems. *Journal of Systems and Software*, 12(2):255–307, 1990.

18. Mike Gordon. A classical mind: Essays in honour of C.A.R. Hoare. In Anthony W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter A Mechanized Hoare Logic of State Transitions, pages 143–159. Prentice Hall, 1994.
19. Volkmar H. Haase. Real-time behavior of programs. *IEEE Transactions on Software Engineering*, 7(5):594–501, September 1981.
20. Ghaith Haddad, Faraz Hussain, and Gary T. Leavens. The design of SafeJML, a specification language for SCJ with support for WCET specifications. In *Proceedings of JTRES'10*, pages 155–163. ACM, 2010.
21. Reinar Hähnle and Einar Broch Johnsen. Resource-aware applications for the cloud. *IEEE Computer*, May 2015. To appear.
22. Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
23. Jozef Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–826, 1994.
24. Jozef Hooman. Compositional verification of real-time applications. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Compos '97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 276–300. Springer, 1998.
25. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
26. Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Proc. Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 71–86. Springer, November 2012.
27. Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 2014. Available online.
28. Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(2):47–72, 1996.
29. Joan Krone, William F. Ogden, and Murali Sitaraman. Modular verification of performance constraints. In *ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 60–67, 2001.
30. Joan Krone, William F. Ogden, and Murali Sitaraman. Profiles: A compositional mechanism for performance specification. Technical Report RSRG-04-03, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, June 2004.
31. Leslie Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Workshop on Theory of Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. LNCS 736, Springer, 1993.
32. Leslie Lamport. Introduction to TLA. Technical report, SRC Research Center, December 1994. Technical Note.
33. Bertrand Meyer. Design by contract: The Eiffel method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.
34. Carrol C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.
35. Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
36. Hanne Riis Nielson. *Hoare-Logic for Run-Time Analysis of Programs*. PhD thesis, Edinburgh University, 1984.

37. Hanne Riis Nielson. A Hoare-like proof-system for run-time analysis of programs. *Science of Computer Programming*, 9, 1987.
38. Peter Puschner and Alan Burns. A review of worst-case execution time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.
39. Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'06)*, pages 202–211, 2006.
40. Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.
41. Mary Shaw. A formal system for specifying and verifying program performance. Technical Report CMU-CS-79-129, Carnegie Mellon University, June 1979.
42. Murali Sitaraman. Compositional performance reasoning. In *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, may 2001.
43. Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *ACM Sigsoft Symposium on Software Reuse*, 2001.
44. Yih-Kuen Tsay. Compositional verification in linear-time temporal logic. In Jerzy Tiuryn, editor, *Proceedings of FoSSaCS 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2000.
45. Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.
46. Alan Wills. Capsules and types in Fresco: Program verification in Smalltalk. In Pierre America, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 1991.

Appendix B

Static Analysis of Cloud Elasticity

Static analysis of cloud elasticity*

Abel Garcia Cosimo Laneve Michael Lienhardt

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus

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) codes 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 informations for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioral types and return cost equations, and (*iii*) an automatic off-the-shelf solver for the the cost equations.

We have experimentally evaluated our technique using a cost analysis solver and we report some results. The technique in this paper may be also applied to estimate (heap) memory consumptions in object-oriented languages.

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 facts, 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 sample codes, extrapolating this information for fully real-life complex programs could be

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

cumbersome and highly error-sensitive. The first attempts about the analysis of resource usages dates back to Wegbreit's pioneering work in 1975 [20], 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 [3, 9]) and to make the connection between programs and closed-form expressions precise (see for instance [19, 13]). We postpone the discussion of the related work in the literature 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 [4]. 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? Under the assumption that one can acquire a virtual machine that has been previously released. A solution to this problem 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:

- 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 the cost of methods like `double_release` in a precise way, in Section 4 we associate methods with abstract descriptions that also carry informations about parameter states and their identities. These descriptions are called *behavioural types* and are formally connected to the programs by means of a typing system.

In Section 5 we therefore analyse behavioural type descriptions by translating them in codes that are adequate for a powerful off-the-shelf solver – the `CoFloCo` solver [10]. 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 facts, 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 the two arguments equal then its cost cannot be computed by 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.

In Section 7 we have we report the results of some of our experimental evaluation. In particular, we compute the cost of `double_release` and two implementation of the factorial functions by means of `CoFloCo`.

Our technique target a simple concurrent language with explicit operations of creation and release of resources. The language is defined in Section 2 and we discuss restriction that ease the development of our technique in Section 3. We discuss how these restriction can be removed and outline our correctness proof in Section 6. We deliver concluding remarks in Section 9.

In this paper we use the metaphor of cloud computing and virtual machines. We observe that our technique actually addresses every type of resources that retain operations of acquire (or creation) and release, such as heap usages in concurrent object-oriented languages.

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<Int>`. 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:

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

A (*pure*) expression e is either an integer constant p, or a variable x, or the reserved identifier `this`, or the standard arithmetic, relational and boolean operations. Since our analysis will be parametric with respect to the inputs, we will parse expressions in a careful way. In particular we split them into *size expressions* se, which are expressions in presburger arithmetics, and *non-size expressions* nse, which are the other type of expressions. The syntax of size and non-size expressions is the following:

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

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 `get` operation. Operations `get` are *not blocking*: 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. Operations taking place on different virtual machines may execute in parallel, while operations in the same virtual machine interleave their evaluation.

A statement s may be either one of the standard operations of an imperative language or the `release` operation. The operation `release`(x) marks the virtual machine x for disposal. Method invocations performed by a released machine, as well as, creations and releases of machines, always return erroneous values.

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 that `return` statements have no continuation.

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

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

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', $\cdots$. We assume there are infinitely many vm names and future names. 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 $\{l \mid \varepsilon\}$, 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 value of f has still to be computed.
- *method invocations* $invoc(o, f, \mathtt{m}, \overline{v})$.

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 either in l. 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 $[\![f]\!]_l = f$.
- $\mathrm{bind}(o, f, \mathtt{m}, \overline{v}) = \{[\mathrm{destiny} \mapsto f, \overline{x} \mapsto \overline{v}] \mid s\{^o/_{\mathtt{this}}\}\}$, 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 p. We focus on rules concerning the method invocations and the management of virtual machines in `vml`, since the other ones are standard.

(New-VM) creates a virtual machine and makes it active. 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 $\bot$.

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

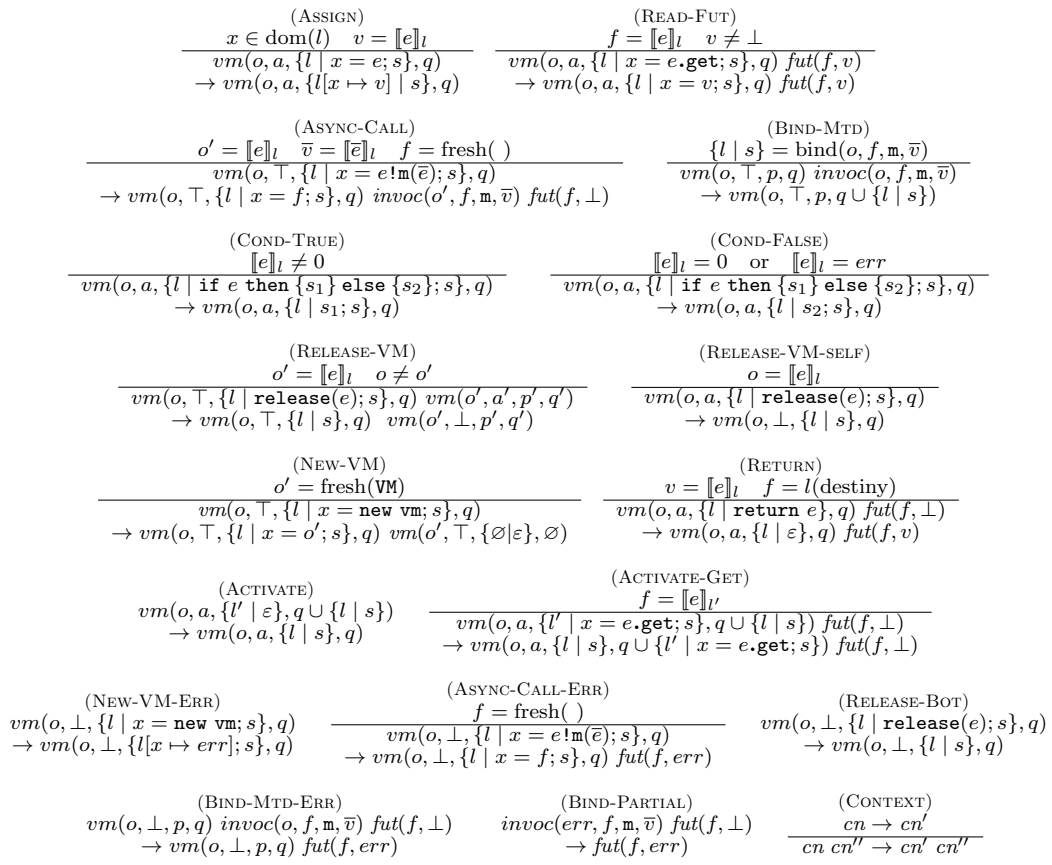

Figure 1: Semantics of `vml`.

(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

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

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, in order to be executed, `costly_fact` needs `n` virtual machines (in addition to the one where the method is performed).

The analysis of `costly_fact` has been easy because the `release` operation carries a locally created virtual machine. Yet, in `vml`, `release` may also apply to method arguments and this is the major source of difficulties for the analysis. Consider for instance the following code:

```
Int first_method() {
  VM x ; Fut<Int> f ; Fut<Int> g ;
  x = new vm;
  f = x!unknown_method(this) ; f.get ;
  g = x!second_method() ; g.get ;
  release x ;
  return 0 ; }
```

A rough analysis might indicate that `first_method` creates a virtual machine, invokes two methods on that machine, and then releases it. However, this analysis is wrong when `unknown_method` releases its argument(s). For instance, if `unknown_method` releases the argument `this`, the invocation of `second_method` and the statement `release x` will not be executed. In order to let the cost analysis be compositional, we record the effects of methods on virtual machines in the arguments and we compute the cost analysis accordingly. One might argue that compositionality might be achieved, in this case, by admitting an over- or under-approximate output instead of recording method's effects on arguments. Actually these approximations are not reasonable if one does not consider releases on the arguments:

- an over-approximation might return very imprecise results. Consider, for instance the case when the invocation `x!unknown_method(this)` releases the two arguments `x` and `this`. The over-approximation will compute the cost of `second_method`, even if it will never be called. This means that the

analysis might output a very large cost because second_method creates a large number of virtual machines.

– an under-approximation might return erroneous results, as it will consider that `release x` will be executed (corresponding to a `-1` cost) while we actually have a null cost.

3 Problems, solutions and restrictions

In this section, we present the two new important concepts linked to resource removal, their properties, and which restrictions we put on input programs to keep our analysis from being too complex (indeed, most of the restriction presented here can be relaxed by increasing the complexity of our analysis; this will be discussed in Section 9).

Method's effects. As discussed in the foregoing example `first_method`, in order to augment the precision of the cost analysis and to avoid erroneous outputs, our analysis records the effects that a method has on its interface. To keep the formalism as simple as possible, we restrict these effects to be *a set of virtual machines in method's interface*, which is noted `R` in Section 4. This simplicity – `R` is a set – has a price: the virtual machines in the interface of a method that will be released in *every execution path* is always the same. That is, a method as

```
Int ugly_release(VM x1, ... , VM xn) {
  VM x ; Fut<Int> f ;
  x = new vm ; f = x!release_all(x1, ..., xn) ; release x ; f.get ;
  return 0; }
```

cannot be handled by our analysis (`release_all(x1, ..., xn)` disposes the virtual machines `x1, ..., xn`). In facts, since `release x` is performed before the synchronisation with `release_all` – statement `f.get` –, the method `release_all` can be stopped at any point of its execution, thus making the set of released virtual machines non-determinate. In order to ban methods like `ugly_release`, we constrain definitions as follows:

1. every method invocation is synchronized within caller's body. In this way every effect of a method is computed before its termination.

2. it is not possible to release a machine that is executing a method.

Virtual machines' identity. Removals of virtual machines may have side effects on other machines. The following method `double_release` illustrates the point:

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

```
}
```

This method takes two machines and simply releases them. This means that, when the virtual machines in input are active *and different*, the cost of `double_release` is `-2`. In fact, this is the case of the following method `user1`:

```
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 ; }
```

An easy computation returns a (net) cost of `0` for `user1`. However, the cost of `double_release` is *not always* `-2`. For example, consider the `user2` above. This method creates one virtual machine (cost `+1`) and invokes `double_release` with a duplicated argument: in this case the cost of `double_release` is `-1`, not `-2`, (and the cost of `user2` is `0`). Said more explicitly, the cost of a method depends on the identity of its arguments, not only on their states (alive or dead).

It is also worth to notice that the identity of arguments has impact on method's effects. Consider the following code snippet:

```
VM x = new vm; VM y = new vm;
x!double_release(x,y);
```

This code creates two new virtual machines and releases them by invoking `double_release`. The awkward point is that the callee machine coincides with the first argument. Therefore `double_release` will fail to release the second argument `y`.

In order to comply with these issue we annotate every operation that may have side effects with *identity sharing informations*, noted Θ in Section 4 – and generate a different cost function for every Θ. For instance, we have two (net) cost functions for the `double_release` method:

1. one where we consider `x` and `y` to be different: here its cost will be `-2` when the two machines are alive,
2. one where se consider the two parameters to be equal: here its cost will be `-1` when the two machines are alive.

In addition, in order to avoid the problematic issue of the code above, we forbid the callee machine to have the same identity of the (other) arguments of the invocation.

Release of carrier's machine. To simplify our analysis, we admit releases of the `this` machine – the carrier – to be the last statement (before `return`). While this restriction may be easily dropped, it avoids duplications of rules in Figure 3 to deal with the fact that the virtual machine `this` could be dead. In any case, we notice that this constraint does not affect the expressive power of the language.

4 The behavioral type system of vml

Our analysis uses abstract descriptions, called *behavioural types*, which are intermediate codes highlighting the features of vml programs that are relevant for the analysis of resources in Section 5. These types support compositional reasonings and are associated to programs by means of a type system.

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 the following

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

Behavioural types express creations of virtual machines ($\nu\alpha$) and their removal ($\alpha^{\checkmark}$), method invocations ($\nu f : \mathtt{m}\ \alpha(\overline{\mathbb{s}})$) and corresponding retrieval of the value ($f^{\checkmark}[\Theta, \mathtt{R}]$) and the conditionals (respectively $(se)\{\mathbb{c}\} + (\neg se)\{\mathbb{c}'\}$ or $\mathbb{c} + \mathbb{c}'$, according to the boolean guard is a size expressions that depends on the arguments of a method or not). Behavioural types also carry *vm name environments* Θ, Θ', $\cdots$. These environments map vm names to extended values $\mathtt{F}\mathbb{t}$, which are called *vm states* in the following. This feature is new in behavioural types and, as discussed in Section 3, it is needed during the analysis to manage the identity of methods' arguments. We will provide additional examples in the rest of the paper to explain vm name environments and their use.

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 vm states. These environments are used in the typing proofs and are dropped in the final types (method types and the main statement type).

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 $x \notin \mathrm{dom}(\Gamma)$. Moreover we define

$$\Gamma[x \mapsto \mathbb{x}](y) \stackrel{def}{=} \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases} \qquad \Gamma|_X(x) \stackrel{def}{=} \begin{cases} \Gamma(y) & \text{if } x \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

The following operation and notations are going to be used:

- vm values $\mathbb{t}$ are partially ordered by the relation $\leq$ defined by $\partial \leq \top$ and $\partial \leq \bot$. 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 $\partial \sqcap \alpha$ is not defined.
- the *update of a vm name environment* Θ with respect to f and Γ (we remind that Θ is defined on vm names only), written $\Theta \searrow^f \Gamma$, returns a vm name environment defined as follows:

$$\Theta \searrow^f \Gamma \stackrel{def}{=} [\alpha \mapsto \quad (\mathtt{F}' \setminus \{f\})(\mathbb{t} \sqcap \mathbb{t}')]^{\alpha \in \mathrm{dom}(\Theta), \Theta(\alpha) = \mathtt{F}\mathbb{t}, \Gamma(\alpha) = \mathtt{F}'\mathbb{t}'}$$

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

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

  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 systems for expressions, expressions with side effects and statements are reported in Figures 2 and 3. It is worth to notice that the type system

$$\text{(T-Var)}\ \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(T-Primitive)}\ \Gamma \vdash \mathtt{p} : \mathtt{p} \qquad \text{(T-Op)}\ \frac{\Gamma \vdash e_1 : se_1 \qquad \Gamma \vdash e_2 : se_2}{\Gamma \vdash e_1\ \mathtt{op}'\ e_2 : se_1\ \mathtt{op}'\ se_2} \qquad \text{T-Unit}\ \frac{\Gamma \vdash e : se}{\Gamma \vdash e : \_}$$

$$\text{(T-Op-Unit)}\ \frac{\Gamma \vdash e_1 : \_ \quad \mathtt{or} \quad \Gamma \vdash e_2 : \_ \quad \mathtt{or} \quad \mathtt{op} \in \{*, /\}}{\Gamma \vdash e_1\ \mathtt{op}\ e_2 : \_} \qquad \text{(T-Pure)}\ \frac{\Gamma \vdash e : \mathbb{x}}{\Gamma \vdash e : \mathbb{x}, 0 \triangleright \Gamma}$$

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

Figure 2: Typing rules for expressions

for expressions 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. As regards the rules for statements, we discuss (T-Invoke), (T-Get), (T-Release), and (T-New) because the other ones are straightforward.

(T-Invoke)

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

(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-New)

$$\frac{\beta \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} \mathtt{new\ vm} : \beta, \nu\beta[\Gamma|_{\mathtt{S}}] \triangleright \Gamma[\beta \mapsto \varnothing\top]}$$

(T-Get)

$$\frac{\begin{array}{c}\Gamma \vdash x : f \qquad \Gamma \vdash f : (\mathtt{o}, \alpha, \Theta, \mathtt{R}) \\ \Theta' = \Theta \overset{f}{\searrow} \Gamma \qquad \mathtt{R}' = fv(\mathtt{o}) \setminus (\mathtt{R} \cup \mathrm{dom}(\Theta)) \\ \Gamma' = \Gamma[\beta \mapsto \varnothing\bot]^{\beta \in \mathtt{R}}[\beta' \mapsto \Theta'(\alpha)]^{\beta' \in \mathtt{R}'}[f \mapsto \mathtt{o}]\end{array}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathtt{o}, f^{\checkmark}\,[\Theta', \mathtt{R}] \triangleright \Gamma'}$$

(T-Get-Done)

$$\frac{\Gamma \vdash x : f \qquad \Gamma \vdash f : \mathtt{o}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathtt{o}, \mathtt{0} \triangleright \Gamma}$$

(T-Release)

$$\frac{\Gamma \vdash x : \alpha}{\Gamma \vdash_{\mathtt{S}} \mathtt{release}(x) : \alpha^{\checkmark}\,[\Gamma|_{\mathtt{S} \cup \{\alpha\}}] \triangleright \Gamma[\alpha \mapsto \varnothing\bot]}$$

(T-Assign-Var)

$$\frac{\Gamma(x) = \mathtt{x} \qquad \Gamma \vdash_{\mathtt{S}} z : \mathtt{x}', \mathtt{c}}{\Gamma \vdash_{\mathtt{S}} x = z : \mathtt{c}[x \mapsto \mathtt{x}']}$$

(T-If)

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

(T-If-ND)

$$\frac{\Gamma \vdash e : \_ \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathtt{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathtt{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathtt{c}_1 + \mathtt{c}_2}$$

(T-Seq)

$$\frac{\Gamma \vdash_{\mathtt{S}} s_1 : \mathcal{C}[\mathtt{a}_1 \triangleright \Gamma_1] \cdots [\mathtt{a}_n \triangleright \Gamma_n] \qquad \Gamma_i \vdash_{\mathtt{S}} s_2 : \mathtt{c}'_i}{\Gamma \vdash_{\mathtt{S}} s_1; s_2 : \mathcal{C}[\mathtt{a}_1 \fatsemi \mathtt{c}'_1] \cdots [\mathtt{a}_n \fatsemi \mathtt{c}'_n]}$$

(T-Return)

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathtt{o} \qquad \Gamma \vdash \mathtt{destiny} : \mathtt{o}' \\ \mathtt{o} \in \mathtt{S} \cup \{\_\} \quad \text{if and only if} \quad \mathtt{o} = \mathtt{o}'\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{return}\ e : \mathtt{0} \triangleright \Gamma[\mathtt{destiny} \mapsto \mathtt{o}]}$$

Figure 3: Type rules for expressions with side effects and statements.

Rule (T-Invoke) types method invocations $e!\mathtt{m}(\overline{e})$ by means of a new 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, a vm name environment and the set of vm names that the method is going to remove. The vm name environment records the state of vm names when the method is invoked and it will be used when the method is synchronized to update the environment of the synchronisation (see rule (T-Get)) with the changes performed by methods in parallel. It is important to observe that the environment returned by the judgment is updated with informations 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, 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 subclases, 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 because the nondeterminism of the execution makes the analysis too much imprecise (in our view). In facts, as discussed in Section 3, in case (*ii.a*), we cannot determine what subset of $\mathtt{R}$ will be actually released (because of the parallel method releasing α). 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. It is worth to notice that the constraint $\mathtt{R} = \varnothing$ of case (*ii*) enforces a programming style that reduces uncertainty and provides a more precise cost analysis. The constraint in the third line of the premise of (T-Invoke) applies the same constraint of the callee to the other invocations in parallel and to the object executing $e!\mathtt{m}(\overline{e})$: it is not possible to remove a vm name that is a callee of a parallel method.

Rule (T-Get) defines the synchronisation with a method invocation that corresponds to a future f. Let $(\mathfrak{o}, \alpha, \Theta, \mathtt{R})$ 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 it is a virtual machine that has been created by the method of f. This state is the same of the callee vm name (which may have been updated since the invocation), namely the value of $(\Theta \searrow^f \Gamma)(\alpha)$. As regards the behavioural type $f^{\checkmark}[\Theta \searrow^f \Gamma, \mathtt{R}]$ of $x.\mathtt{get}$, it carries two arguments (*i*) the vm name environment of the invocation updated with the invocation of parallel methods and (*ii*) the vm names to be released. These two arguments will let us disable the removal of names in the cost analysis when these names are removed by methods in parallel. In this case, the removal is computed in the caller method (and therefore it counts `-1`, instead of `-1` in every method in parallel) – see definition of `translate` in case of $f^{\checkmark}[\Theta \searrow^f \Gamma, \mathtt{R}]$ in Section 5.

Rule (T-release) models the removal of a vm name α. Notice that the behavioural type is not just $\alpha^{\checkmark}$, which should correspond to a `-1` in the cost analysis. Rather, it is a "conditional" removal because the name may be removed by some method in parallel, or even may have been already removed when the method has been invoked. This is the reason for the presence of the vm name environment $[\Gamma|_{\mathtt{S}\cup\{\alpha\}}]$. We also observe that this rule applies only when the caller has not been already released.

For reasons similar to the rules discussed above, (T-New) adds an environment $\Gamma|_{\mathtt{S}}$ to the type $\nu\beta$ expressing a new resource. In facts, this environment will make the cost increase by `1` provided the caller method has not been already released – see definition of `translate` in case of $\nu\beta[\Gamma|_{\mathtt{S}}]$ in Section 5.

The type system of `vml` is completed with the rules for method declarations and programs:

(T-Method)

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

(T-Program)

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

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 $\circledcirc$ may be either $\_$ or a vm name in `S` or a fresh vm name. In this last case, the premise $(\Gamma_i(\gamma) = \Gamma_j(\gamma))^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup fv(\circledcirc)}$ guarantees that every branch in the returned behavioural type creates a new vm name and, by rule (T-Return), the chosen vm name must be always the same.

We display behavioural types at work by typing codes of Section 2 and 3. Actually, the following types do not abstract a lot from codes' details 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 α(n) {
  (n==0){ 0 }
  + (n>0){ νy : fact α(n − 1) ;
           y✓[α ↦ ∅α] ; }
} - , { }
```

```
costly_fact α(n) {
  (n==0){ 0 }
  + (n>0){ νβ[α ↦ ∅α] ;
           νx : costly_fact β(n − 1) ;
           x✓[α ↦ ∅α, β ↦ ∅⊤] ;
           β✓[α ↦ ∅α, β ↦ ∅⊤] ; }
} - , { }
```

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 types of `double_release` and `user1` are the following ones

```
double_release α(β, γ) {
  β✓[α ↦ ∅α, β ↦ ∅β, γ ↦ ∅γ] ;
  γ✓[α ↦ ∅α, β ↦ ∅⊥, γ ↦ ∅γ] ;
} - , {β, γ}
```

```
user1 α( ){
  νβ[α ↦ ∅α] ; νγ[α ↦ ∅α] ;
  νf : double_free α(β, γ) ;
  f✓[α ↦ ∅α, β ↦ ∅⊤, γ ↦ ∅⊤] ;
} - , { }
```

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. In facts, in case of `user1` these values are $\varnothing\top$ and this will mean that the cost of $f^{\checkmark}[\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\top, \gamma \mapsto \varnothing\top]$ will be `-2`.

5 The analysis of behavioural types

The behavioural types returned by system in Section 4 are used to compute the resource elasticity of a `vml` program. This computation is performed by an off-the-shelf solver – the `CoFloCo` solver [10] – and, in this section, we discuss the translation of a behavioural type program into a set of *recurrence relations* that fed the solver.

Basically, our translation maps method types into functions from parameters to cost, where

- method invocations are translated into function calls,
- 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 `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 `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 that pertain to method invocations: the management of arguments' identities and of argument's values.

Argument's identities. Consider the code

```
Int free() { release(this) ; return(0) ; }

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

The behavioural types of these methods are

$$\mathtt{free}\ \alpha(\,)\{\ \alpha^{\checkmark}[\alpha \mapsto \varnothing\top]\ \} \_, \{\alpha\}$$
$$\mathtt{m}\ \alpha(\beta,\gamma)\{\nu\, f : \mathtt{free}\,\beta(\,) \fatsemi \gamma^{\checkmark}[\Theta] \fatsemi f^{\checkmark}[\Theta']\} \_, \{\beta,\gamma\}$$

where

$$\Theta = \alpha \mapsto \varnothing\alpha, \beta \mapsto \{\mathtt{free}\}\beta, \gamma \mapsto \varnothing\gamma$$
$$\Theta' = \alpha \mapsto \varnothing\alpha, \beta \mapsto \{\mathtt{free}\}\beta, \gamma \mapsto \varnothing\bot$$

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 facts, 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 $\texttt{free}(\beta)$ and therefore it counts `0`. To solve this problem of arguments' identity, we refine even more the translation of method, which now depends on an equivalence relation telling which of the vm names in parameter are actually equal or not. Hence, in general, our method `m` will be translated in four cost functions: $\texttt{m}^{\{1\},\{2\}}_{\text{peak}}(x,y)$ and $\texttt{m}^{\{1\},\{2\}}_{\text{net}}(x,y)$, which correspond to the invocations where $x \neq y$, and $\texttt{m}^{\{1,2\}}_{\text{peak}}(x)$ and $\texttt{m}^{\{1,2\}}_{\text{net}}(x)$, which correspond to the invocations where $x = y$. (The equivalence relation in the superscript never mention `this`, which is also an argument, because we assume that `this` is always different from 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:

$$\texttt{EqRel}(\alpha_1, \cdots, \alpha_n) \;=\; \bigcup_{i \in 1..n} \{\, \{j \mid \alpha_j = \alpha_i\} \,\}$$

Let $\texttt{EqRel}(\alpha_1, \cdots, \alpha_n)(\beta_1, \cdots, \beta_n)$ be the tuple $(\beta_{i_1}, \cdots, \beta_{i_k})$, where $i_1, \cdots, i_k$ are *canonical representatives* of the sets in $\texttt{EqRel}(\alpha_1, \cdots, \alpha_n)$ (we take the vm name with the least index in every set). We observe that, by definition, $\texttt{EqRel}(\alpha_1, \cdots, \alpha_n)(\alpha_1, \cdots, \alpha_n)$ is a tuple of pairwise different vm names (in $\alpha_1, \cdots, \alpha_n$).

We will always assume that $\{1\} \in \texttt{EqRel}(\alpha_1, \cdots, \alpha_n)$. Since α_1 always represents the `this` object, this constraint enforces that the other arguments of invocations are always different from `this` (see the discussion in the paragraph about virtual machines identity in Section 3).

Argument's values. Consider the code

```
Int m(VM x) {
  Fut<Int> f ; Fut<Int> g ; VM y ; VM z ; Int u ;
  y = new vm ; z = new vm ;
  f = x!free() ; g = y!foo(x, z) ;
  u = g.get ; u = f.get ; return 0 ;
}
```

where `free` is as above and `foo` is unspecified (we assume that `foo` does not release any machine). The behavioural types of `m` is

$$\texttt{m}\;\alpha(\beta)\{\; \nu\gamma[\Theta_\gamma] \;;\; \nu\gamma'[\Theta_{\gamma'}]; \nu\, f : \texttt{free}\,\beta(\;) \;;\; \nu\, g : \texttt{foo}\,\gamma(\beta,\gamma') \;;\; g^{\checkmark}[\Theta_2]; f^{\checkmark}[\Theta_1]\}^{\;-\;,\{\beta\}}$$

where

$$\begin{array}{rcl} \Theta_\gamma & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta] \\ \Theta_{\gamma'} & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta, \gamma \mapsto \varnothing\top] \\ \Theta_2 & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \{f\}\beta, \gamma \mapsto \varnothing\top, \gamma' \mapsto \varnothing\top] \\ \Theta_1 & = & [\alpha \mapsto \varnothing\alpha, \beta \mapsto \{f\}\beta, \gamma \mapsto \varnothing\top, \gamma' \mapsto \varnothing\top] \end{array}$$

The arguments of the invocation of `foo` are β and γ' and, in order to compute its cost, it is necessary to instantiate these arguments with actual vm values. This instantiation is straightforward for γ': there is no concurrent future releasing it, so its state is $\top$. However, this is not the case for β': as specified in Θ_2, its vm state is tagged with a nonempty set of futures because `free` is concurrently releasing it. Hence, we need to translate this state into a simple one ($\bot$, $\top$, or ∂). In facts, the value of the first argument of `foo` may be ∂ if β was originally either $\top$ or ∂, or it may be $\bot$ if β was already released. To deal with this case, we introduce an operator on vm values $\beta\downarrow$ whose meaning is the one just described:

$$\beta\downarrow = \begin{cases} \partial & \text{if } \beta = \top \text{ or } \beta = \partial \\ \bot & \text{else} \end{cases}$$

Since it is not possible to know the value of β during the translation, we use $\beta\downarrow$ in the syntax of our cost expressions. This term will be evaluated during the solving process.

To conclude, the translation of vm states into vm values (extended with $\alpha\downarrow$), written $\mathtt{F}\mathbb{t}\Downarrow$, is defined as follows:

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

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 translation function. The translation function, called `translate`, is structured in three parts that respectively correspond to simple atoms, full behavioral 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 (unsynchronized);

3. α is the vm name of the virtual machine of the current behaviour type;

4. $(se)\{\,\overline{e}\,\}$ is sequence of costs of the current execution branch: the size expression se stores all the conditions corresponding to the current execution branch, while $\overline{e}$ is the sequence of (over-approximated) costs that branch takes during its execution.

5. the behavioural type that is translated; it may be either $\mathbb{a}$, $\mathbb{c}$ or $\overline{\mathbb{C}}$.

The auxiliary functions below let us reduce the number of cases of the definition of `translate`:

$$\texttt{CNEW}(\alpha) \;=\; \begin{cases} \texttt{0} & \alpha = \bot \\ \texttt{1} & \text{otherwise} \end{cases}$$

This 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.

$$\texttt{CREL}(\alpha) \;=\; \begin{cases} \texttt{-1} & \alpha = \top \\ \texttt{0} & \text{otherwise} \end{cases}$$

This 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 ($\alpha = \top$).

The `translate` function also uses the *merge operation*, noted $\Theta[\Delta]$, that takes a vm name environment Θ and an equivalence relation on vm names Δ and returns a *substitution*. We remind that vm name environments have been introduced in Section 4 to manage identities of arguments in method calls. Take, for instance, the atom $f^{\checkmark}[\Theta, \texttt{R}]$ within a behavioural type that binds f to $\texttt{foo}\,\alpha(\beta,\gamma)$. Assume to evaluate this behavioural type for $\texttt{m}^{\Delta}_{\text{peak}}$ where $\Delta = \{\beta, \gamma\}$. That is, the two arguments are actually identical. What are the values of β and γ for evaluating $\texttt{foo}^{\Delta}_{\text{peak}}$ and $\texttt{foo}^{\Delta}_{\text{net}}$? 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 $\Theta(\beta)$ and $\Theta(\gamma)$ (but greater than any other value that is smaller);
3. to substitute β and γ with the result of 2.

For example, let $\Theta = [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta, \gamma \mapsto \varnothing\gamma]$ and $\Delta(\beta) = \Delta(\gamma) = \beta$. We expect that a value for the item 2 above is $\varnothing\beta$ and the substitution of the item 3 is $\{^{\varnothing\beta,\,\varnothing\beta}/_{\beta,\gamma}\}$. Formally, the operation returning the value for 2 is noted $\otimes^{\beta}$ below (it applies to vm values and vm states) and the operation returning the substitution of item 3 is the merge operation.

$$\mathtt{t} \otimes^{\alpha} \mathtt{t}' \;=\; \begin{cases} \bot & \text{if } \mathtt{t} = \bot \text{ or } \mathtt{t}' = \bot \\ \alpha\downarrow & \text{if } (\mathtt{t} = \gamma\downarrow \text{ and } \mathtt{t}' \neq \bot) \text{ or } (\mathtt{t} \neq \bot \text{ and } \mathtt{t}' = \gamma\downarrow) \\ \alpha & \text{if } \mathtt{t} = \gamma \text{ and } \mathtt{t}' = \gamma' \end{cases}$$

$$\texttt{F}_1\mathtt{t}_1 \otimes^{\alpha} \texttt{F}_2\mathtt{t}_2 \;=\; (\texttt{F}_1 \cup \texttt{F}_2)(\mathtt{t}_1 \otimes^{\alpha} \mathtt{t}_2)$$

$$\Theta[\Delta] \;:\; \left[\alpha \mapsto \bigotimes^{\Delta(\alpha)} \{\Theta(\beta) \mid \beta \in \mathrm{dom}(\Theta) \text{ and } \Delta(\beta) = \Delta(\alpha)\}\right]^{\alpha \in \mathrm{dom}(\Theta)}$$

We notice that the definition of $\otimes^{\alpha}$ is not necessary for vm values as ∂ or $\top$ because we merge vm names whose image by Θ can only be either $\mathtt{F}\beta$ or $\mathtt{F}\beta\downarrow$ or $\mathtt{F}\bot$. As a notational remark, we observe that the substitution $\Theta[\Delta]$ is noted as a map $[\alpha_1 \cdots \mathtt{F}_1\mathbb{t}_1, \cdots, \alpha_n \cdots \mathtt{F}_n\mathbb{t}_n]$ 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 $\Theta[\Delta](\alpha)$ or even $\Theta[\Delta](\alpha_1, \cdots, \alpha_n)$ with the obvious meanings.

Every preliminary notion is in place for defining `translate`. We begin with the translation of atoms.

$$
\begin{array}{l}
\mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}; e\,\}, \mathbb{a}) \;=\; \\
\quad \begin{cases}
(\Psi, (se)\{\,\overline{e}; e\,\}) & \text{when } \mathbb{a} = \mathtt{0} \\
(\Psi, (se)\{\,\overline{e}; e; e + \mathtt{CNEW}(\mathbb{t})\,\}) & \text{when } \mathbb{a} = \nu\beta[\Theta] \\
 & \text{and } \Theta[\Delta](\alpha) = \mathtt{F}\mathbb{t} \\
(\Psi, (se)\{\,\overline{e}; e; e + \mathtt{CREL}(\mathbb{t})\,\}) & \text{when } \mathbb{a} = \beta^{\checkmark}[\Theta] \\
 & \text{and } \Theta[\Delta](\beta) = \mathtt{F}\mathbb{t} \\
(\Psi[f \mapsto \mathtt{m}\ \Delta(\beta)(\overline{se}, \Delta(\overline{\beta}))], (se)\{\,\overline{e}; e; e + f\,\}) & \text{when } \mathbb{a} = (\nu f = \mathtt{m}\ \beta(\overline{se}, \overline{\beta})) \\
(\Psi \setminus f, (se)\{\,(\overline{e}; e)\sigma; (\overline{e}; e)\sigma' + \sum_{\gamma \in \Delta(\mathtt{R}), \Theta[\Delta](\gamma) = \mathtt{F}\mathbb{t}, \mathtt{F} \neq \varnothing} \mathtt{CREL}(\mathbb{t})\,\} & \text{when } \mathbb{a} = f^{\checkmark}[\Theta, \mathtt{R}] \\
 & \text{where} \\
 & \Psi(f) = \mathtt{m}\ \beta(\overline{se}, \overline{\beta}) \\
 & \mathtt{EqRel}(\beta, \overline{\beta}) = \Xi \\
 & \sigma = \{{}^{\mathtt{m}^{\Xi}_{\mathrm{peak}}(\overline{se}, \Theta[\Delta](\Xi(\beta, \overline{\beta}))\ \Downarrow)}/_{f}\} \\
 & \sigma' = \{{}^{\mathtt{m}^{\Xi}_{\mathrm{net}}(\overline{se}, \Theta[\Delta](\Xi(\beta, \overline{\beta}))\ \Downarrow)}/_{f}\}
\end{cases}
\end{array}
$$

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 $f^{\checkmark}[\Theta, \mathtt{R}]$. In this last case, let $(se)\{\,\overline{e}; e\,\}$ 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 the purpose of the addend $\sum_{\gamma \in \Delta(\mathtt{R}), \Theta[\Delta](\gamma) = \mathtt{F}\mathbb{t}, \mathtt{F} \neq \varnothing} \mathtt{CREL}(\mathbb{t})$ 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 $\mathtt{m}_{\mathrm{peak}}$ and $\mathtt{m}_{\mathrm{net}}$ that are invoked are those corresponding to the equivalence relation of the tuple $(\beta, \overline{\beta})$.

The definition of `translate` for behavioural type is given below. It follows straightforwardly by composing the definitions of the atoms. We also observe that, in this case, the sequences of costs are sets.

$$\texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}) \;=\; \begin{cases} \{(se')\{\,\overline{e'}\,\}\} & \text{when } \mathbb{c}=\mathbb{a}\triangleright\varnothing \text{ and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{a})=(\Psi',(se')\{\,\overline{e'}\,\}) \\ C'' & \text{when } \mathbb{c}=\mathbb{a}\fatsemi\mathbb{c}' \text{ and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{a})=(\Psi',\{(se')\{\,\overline{e'}\,\}\}) \\ & \text{and } \texttt{translate}(\Delta,\Psi',\alpha,(se')\{\,\overline{e'}\,\},\mathbb{c}')=(\Psi'',C'') \\ C'\cup C'' & \text{when } \mathbb{c}=\mathbb{c}_1+\mathbb{c}_2 \text{ and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}_1)=(\Psi',C') \\ & \text{and } \texttt{translate}(\Delta,\Psi,\alpha,(se)\{\,\overline{e}\,\},\mathbb{c}_2)=(\Psi'',C'') \\ C' & \text{when } \mathbb{c}=(se')\{\mathbb{c}'\} \text{ and } \texttt{translate}(\Delta,\Psi,\alpha,(se\wedge se')\{\,\overline{e}\,\},\mathbb{c}')=(\Psi',C') \end{cases}$$

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

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

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

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

and

$$\texttt{translate}(\Delta,\varnothing,\alpha_1,(\texttt{true})\{\,0\,\},\mathbb{c}) = \bigcup_{i=1}^{n}(se_i)\{\,e_{1,i};\ldots;e_{h_i,i}\,\}$$

Then

$$\texttt{translate}(\Xi,\texttt{m}\;\alpha_1(\overline{x},\alpha_2,\ldots,\alpha_k)\;\{\,\mathbb{c}\,\}:\mathbb{o},\;\texttt{R}) \;=\; \begin{cases} \texttt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=0 & [\alpha_1=\bot] \\ \texttt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{1,1} & [se_1\wedge\alpha_1\neq\bot] \\ \quad\vdots & \quad\vdots \\ \texttt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{h_1,1} & [se_1\wedge\alpha_1\neq\bot] \\ \texttt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{1,2} & [se_2\wedge\alpha_1\neq\bot] \\ \quad\vdots & \quad\vdots \\ \texttt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{h_n,n} & [se_n\wedge\alpha_1\neq\bot] \\ \texttt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=0 & [\alpha_1=\bot] \\ \texttt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=\texttt{m}^{\Xi}_{\text{peak}}(\overline{x},\Xi[\alpha_1,\ldots,\alpha_n]) & [\alpha_1=\partial] \\ \texttt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{n_1,1} & [se_1\wedge\alpha_1=\top] \\ \quad\vdots & \quad\vdots \\ \texttt{m}^{\Xi}_{\text{net}}(\overline{x},\Xi[\alpha_1,\alpha_2,\ldots,\alpha_k])=e_{n_n,n} & [se_n\wedge\alpha_1=\top] \end{cases}$$

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

$$\texttt{translate}(\mathbb{C}_1\;\ldots\;\mathbb{C}_n,\;\mathbb{c}) \;=\; \begin{cases} \texttt{translate}(\mathbb{C}_1)\;\cdots\;\texttt{translate}(\mathbb{C}_n) & \\ \texttt{main}()=e_{1,1} & [se_1] \\ \quad\vdots & \quad\vdots \\ \texttt{main}()=e_{h_1,1} & [se_1] \\ \texttt{main}()=e_{1,2} & [se_2] \\ \quad\vdots & \quad\vdots \\ \texttt{main}()=e_{h_m,m} & [se_m] \end{cases}$$

We show the output of `translate` when applied to the behavioural types of `double_release`, `user1`, and `user2` that we have described in Section 4 (well, the behavioural type of `user2` has not been shown: it is left as an exercise). Since `double_release` has two arguments, we generate two sets of equations, as discussed above. On the contrary, methods `user1` and `user2` carry one argument and therefore there is one set of equations only. 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.

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

$$\begin{cases}
\mathit{double_release}_{\mathrm{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\mathrm{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\mathrm{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\mathrm{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 \neq \bot] \\
\\
\mathit{double_release}_{\mathrm{net}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\mathrm{net}}(\alpha_1,\alpha_2,\alpha_3) = \mathit{double_release}_{\mathrm{peak}}(\alpha_1,\alpha_2,\alpha_3) & [\alpha_1 = \partial] \\
\mathit{double_release}_{\mathrm{net}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 = \top]
\end{cases}$$

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

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

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

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

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

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

6 Outline of the proof of correctness

The proof of correctness of our technique is long even if almost standard (see [11] 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 (the types of cn and cn' are equal) because our types are "behavioural". However, it is critical for the correctness of the cost analysis that there is 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. We also observe that the number of alive virtual machines in a configuration is identical to the direct cost of the corresponding a behavioural type. Then it is also necessary to define

3. the extension of the function `translate` to compute the recurrence relations for behavioural types of configurations. These recurrence relations let us 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 recurrence relations in [2]. This lets us to circumvent possible errors in implementations of the theory, such as `CoFloCo` [10] or `PUBS` [2]. 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* `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 $cn \rightarrow cn'$. 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 and experiments

In this section we discuss technical details about the translation of the recurrence relations in Section 5 into the cost analysis tool we use – the `CoFloCo` analyser [10] – and we examine the outputs obtained for the running examples of this paper. It is worth to notice that instead of targeting `CoFloCo` , we might also target the `PUBS` analyser [2], which also has similar recurrence relations for input.

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 $[\alpha = \texttt{3}]\texttt{3}+[\texttt{1} \leq \alpha \leq \texttt{2}]\texttt{2}$;
- 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 for the cost analyzer is shown in Figure 4 and, in order to evaluate it, we need to specify a so-called *entry point*.

```
eq(doubleRelease123_peak(A,B,C), 0, [], [A = 3]).
eq(doubleRelease123_peak(A,B,C), 0, [], [A < 3]).
eq(doubleRelease123_peak(A,B,C), 0, [crel(B)], [A < 3]).
eq(doubleRelease123_peak(A,B,C), 0, [crel(B), crel(C)], [A < 3]).

eq(doubleRelease123_net(A,B,C), 0, [], [A = 3]).
eq(doubleRelease123_net(A,B,C), 0, [doubleRelease123_peak(A,B,C)], [A = 2]).
eq(doubleRelease123_net(A,B,C), 0, [crel(B), crel(C)], [A = 1]).

eq(user1_peak(A), 0, [], [A = 3]).
eq(user1_peak(A), 0, [], [A < 3]).
eq(user1_peak(A), 0, [cnew(A)], [A < 3]).
eq(user1_peak(A), 0, [cnew(A),cnew(A)], [A < 3]).
eq(user1_peak(A), 0, [cnew(A),cnew(A),doubleRelease123_peak(A, 1, 1)], [A < 3]).
eq(user1_peak(A), 0, [cnew(A),cnew(A),doubleRelease123_net(A, 1, 1)], [A < 3]).

eq(user1_net(A), 0, [], [A = 3]).
eq(user1_net(A), 0, [user1_peak(A)], [A = 2]).
eq(user1_net(A), 0, [cnew(A),cnew(A),doubleRelease123_net(A, 1, 1)], [A = 1]).
```

Figure 4: Cost equations of `double_release` and `user1` in `CoFloCo` format

This entry point has the following format:

```
entry(METHOD_NAME (LIST_OF_ARGUMENTS):[CONDITIONS]).
```

where the first argument always represents the state of the carrier virtual machine. The following table report the output of `CoFloCo` with respect to the entry point. We observe that the computed cost is exactly what we anticipated

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

Table 1: Costs of programs `double_release` and `user1`

in Section 4.

Figure 5 describes the program of `double_release` when used by `user2`. In this case, the arguments of `double_release` are equal, therefore we use $double\_release_{\text{peak}}(\alpha_1, \alpha_2)$ and $double\_release_{\text{net}}(\alpha_1, \alpha_2)$. We write these functions in `CoFloCo` as `doubleRelease12_peak(A,B)` and `doubleRelease12_net(A,B)`.

```
eq(doubleRelease12_peak(A,B), 0, [], [A = 3]).
eq(doubleRelease12_peak(A,B), 0, [], [A < 3]).
eq(doubleRelease12_peak(A,B), 0, [crel(B)], [A < 3]).
eq(doubleRelease12_peak(A,B), 0, [crel(B), crel(3)], [A < 3]).

eq(doubleRelease12_net(A,B), 0, [], [A = 3]).
eq(doubleRelease12_net(A,B), 0, [doubleRelease12_peak(A,B)], [A = 2]).
eq(doubleRelease12_net(A,B), 0, [crel(B), crel(3)], [A = 1]).

eq(user2_peak(A), 0, [], [A = 3]).
eq(user2_peak(A), 0, [], [A < 3]).
eq(user2_peak(A), 0, [cnew(A)], [A < 3]).
eq(user2_peak(A), 0, [cnew(A),doubleRelease12_peak(A, 1)], [A < 3]).
eq(user2_peak(A), 0, [cnew(A),doubleRelease12_net(A, 1)], [A < 3]).

eq(user2_net(A), 0, [], [A = 3]).
eq(user2_net(A), 0, [user2_peak(A)], [A = 2]).
eq(user2_net(A), 0, [cnew(A),doubleRelease12_net(A, 1)], [A = 1]).
```

Figure 5: Cost equations of `double_release` and `user2` in `CoFloCo` format

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. As before, the cost is exactly what we informally

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

Table 2: Costs of programs `double_release` and `user2`

computed in Section 4.

We conclude this section by discussing the cost of the two factorial programs `fact` and `cheap_fact` discussed in Section 2. The list of cost equations generated by `translate` is given in Figure 6 below.

We notice that Figure 6 has a couple of equations commented out. These equations are not allowed as input in `CoFloCo` because they lead to mutually recursive chains, which are banned by the analyser. Yet, in this case, the exclusion of these equations does not affect the result because we assume that the virtual machine executing either `fact` or `costly_fact` is always alive. Table 3 shows the output of `CoFloCo`.

As in the previous examples the *net cost* is equal to `0` in both cases, bacause every created virtual machine is released before the end of the program. In

```
eq(fact_peak(A,B), 0, [], [A = 3]).
eq(fact_peak(A,B), 0, [], [B = 0]).
eq(fact_peak(A,B), 0, [], [B > 0]).
eq(fact_peak(A,B), 0, [fact_peak(1,B)], [B > 0]).
eq(fact_peak(A,B), 0, [fact_net(1,B)], [B > 0]).

eq(fact_net(A,B), 0, [], [A = 3]).
// eq(fact_net(A,B), 0, [fact_peak(A,B)], [A = 2]).
eq(fact_net(A,B), 0, [], [A = 1, B = 0]).
eq(fact_net(A,B), 0, [fact_net(1, B-1)], [A = 1, B > 0]).

eq(costly_fact_peak(A,B), 0, [], [A = 3]).
eq(costly_fact_peak(A,B), 0, [], [B = 0]).
eq(costly_fact_peak(A,B), 0, [], [B > 0]).
eq(costly_fact_peak(A,B), 0, [cnew(A)], [B > 0]).
eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_peak(1, B-1)], [B > 0]).
eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_net(1, B-1)], [B > 0]).
eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_net(1, B-1), crel(1)],
                                                              [B > 0]).

eq(costly_fact_net(A,B), 0, [], [A = 3]).
// eq(costly_fact_net(A,B), 0, [costly_fact_peak(A,B)], [A = 2]).
eq(costly_fact_net(A,B), 0, [], [A = 1, B = 0]).
eq(costly_fact_net(A,B), 0, [cnew(A), costly_fact_net(1, B-1), crel(1)],
                                                    [A = 1, B > 0]).
```

Figure 6: Cost equations of `fact` and `costly_fact` in `CoFloCo` format

Entry Point	Cost
`entry(fact_net(1,B):[B>=0]).`	`0`
`entry(fact_peak(1,B):[B>=0]).`	`0`
`entry(costly_fact_net(1,B):[B>=0]).`	`0`
`entry(costly_fact_peak(1,B):[B>=0]).`	`max([B,1])`

Table 3: Costs of programms `fact` and `cheap_fact`

the case of the *peak cost*, for method `fact` the number of virtual machines will depend on the depth of the recursion, in this case, given by parameter `B`. On the other hand, method `cheap_fact` creates at each step a virtual machine and releases it before the recursive call. This management of virtual machines gives a *peak cost* equal to `1` because it supports the reuse of the released virtual machines.

8 Related Work

After the pioneering work by Wegbreit in 1975 [20] that developed a technique for deriving upper-bounds costs of functional programs, a number of techniques for the cost analysis have been developed. These techniques may be divided

into two categories.

The first category, that we call *classical techniques*, addresses cost analysis in three steps: *(i)* extracting relevant informations out of the original programs, *e.g.* abstracting data structures to their size and assigning a cost to eevery program expression, *(ii)* converting the abstract program into recurrence relations, and *(iii)* solving the cost equations with an automatic tool. Very powerful classical technique tools are [2, 12, 3, 9, 10] that produce very accurate upper bounds expressions for various kinds of programs of different complexity. We refer to [10] for a comparison among some of these techniques. The main drawback of these techniques is the lack of compositionality: it is difficult to scale the analysis to large programs and the translation from the original program to the recurrence relations is always unclear.

The second category, that we call *amortized techniques*, uses the technique based on type systems and amortized analysis developed for functional programs by Hofmann [15]. This approach is highly compositional because of the use of types, and more suitable for formal verification because the connection between the original program and the cost equations can be demonstrated by a standard subject-reduction theorem [15, 19, 16, 17, 13]. We follow this technique in the present work.

A common feature of classical and amortized techniques is that they analyze *cumulative resources*, that is resources that *do not decrease* during the execution of the programs. This is the case, for example, for execution time, number of operations, memory (without an explicit `free` operation), etc. As we discussed, this assumption eases the analysis because it permits to compute over-approximated cost. On the contrary, the presence of *explicit or implicit* release operation entangles the analysis, as already discussed in [6] where a memory cost analysis is proposed for languages with garbage collection. It is worth to say that the scenario of [6] is not difficult because, by definition of garbage collection, released memories are 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*. As discussed in Section 5, the first refers to the worst case cost for an operation to complete, while the second refers to the cost of an operation after completion. For cumulative analysis the two notions coincide; however, 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 Albert *et al.* in [4] have analysed the cost of a language with explicit releases. However, the release operation they consider is used in a very restrictive way: releases can only be performed over locally acquired resources. This constraint ensures having no partially negative costs when analyzing block sequences thus maintaining the restriction of non-negative cost models.

Most cost analysis techniques usually address sequential program. Only few works also address concurrent programs [1, 5, 14]. In this cases, to reduce the imprecision of the analysis caused by the nondeterminism, the authors of [1, 5] use a clever technique for determining parallel codes, called *may-happen-in-parallel* [8]. However no one of these contributions consider a concurrent lan-

guage with a powerful `release` operation that allows one disable the resources taken in input. In facts, 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 informations about resource usages in programs, called behavioural types; an automatic translation that transforms these types into cost expressions; the application of solvers, like `CoFloCo` [10], 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 three lines of work. First, we will complete the technical development of this paper by delivering full proofs of correctness of our technique. Second, 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. Third, we intend to implement our analysis targeting a programming language with a formal model as `ABS` [18].

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and German Puebla. Cost analysis of concurrent OO programs. In *Proceedings of Programming Languages and Systems - APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2011.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán 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.

[3] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[4] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of (TACAS 2015)*, Lecture Notes in Computer Science. Springer, 2015. To appear.

[5] Elvira Albert, Jess Correas, and Guillermo 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] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. *SIGPLAN Not.*, 45(8):121–130, 2010.

[7] Diego Esteban Alonso-Blas and Samir 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] Rajkishore 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] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen 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] Antonio Flores Montoya and Reiner 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.

[11] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling*, 2015. To Appear.

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

[13] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of POPL 2011*, pages 357–370. ACM, 2011.

[14] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs, 2015. [Online; accessed 11-February-2015].

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

[16] Martin Hofmann and Steffen 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.

[17] Martin Hofmann and Dulma 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.

[18] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin 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.

[19] Wei ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin 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.

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