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

Deliverable D1.2.2
Modeling of Resources (Final Report)

Date of document: T0+30

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

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

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:
Modeling of Resources (Final Report)

This document summarises deliverable D1.2.2 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This deliverable describes the outcomes of Task T1.2: The formalization of virtualized resources and their full integration into the syntax and semantics of the abstract behavioral specification language.

The content of this deliverables supercedes Deliverable D1.2.1 (Modeling of Resources, Intermediate Report).

List of Authors

Abel Garcia (BOL)
Samir Genaim (UCM)
Enrique Martín (UCM)
Rudolf Schlatte (UIO)

Contents

Chapter 1

Introduction

Task T1.2 is about the modeling of resources. The task formalizes virtualized resources and integrates them into the syntax and semantics of the ABS language. We aim to introduce virtualized resources as first class citizens of the modeling language, and connect them to concepts of execution and locality.

In order to be realistic, resource management must depend on the execution and data flow in the business code of the object-oriented model, and must reflect the timed behavior of the real system. In this, we rely on the Real-Time ABS extension to the ABS language. This task develops data type descriptions for the different kinds of resources, and considers the generalization from a single resource to a set of resources. Further, we investigate the operational integration of these descriptions in terms of annotations, deployment components and deployment strategies. Finally, to streamline our language definition, we investigate the expressive power of our extensions.

This deliverable describes the results of Task T1.2. This task formalized resource modeling and identified two different uses for resources, which we call *static* and *dynamic*. Static resources form the basis of deployment descriptions, plans and (compile- or run-time) decisions. Dynamic resources form the basis of performance analysis done with simulation tools. This deliverable discusses both, with an emphasis on dynamic resources. The use of static resources is also discussed in the report on Task T1.3 (Modeling of Deployment).

In terms of execution and simulation semantics, this deliverable discusses the three dynamic resource types identified in the context of simulating virtualized systems; i.e., CPU speed, memory size, and bandwidth. We develop data type descriptions for these different resources, and consider the generalization from a single resource to a set of resources. We describe a representation of these resources as first class citizens of the modeling language in order to express load balancing and scheduling decisions for service requests based on resource availability. In terms of deployment strategies, we discuss the static resource types core count and memory size that help Task T1.3 achieve its goals. A discussion of static deployment strategies and resource configurations can be found in [5].

The present deliverable first briefly reviews the language features of real-time ABS that are relevant for resource modeling in Chapter 2, then presents modeling of static and dynamic resources and its influence on the behavior of ABS models in Chapter 3. Finally, Chapter 4 puts the resource models in the context of the static analysis tools developed in Work Packages 2 and 3.

Chapter 2

Real-Time ABS

In order to run timed simulations, which forms the basis of simulations estimating the effect of different resource usage and deployment scenarios, it is necessary to introduce a notion of time into ABS.

The ABS notion of time is *dense time* with *run-to-completion semantics*. This means that in a simulation, most statements conceptually run infinitely fast. Time advances when all processes meet one of the following conditions:

- the cog of the process is blocked
- the process is suspended waiting for time to advance
- the process is waiting for a resource to become available
- the process is suspended or blocked waiting for another process

In practice this means that all process run as long as there is "work to be done." Once all processes are stuck waiting (directly or indirectly) for the clock, the clock advances by an amount sufficient to unblock one or more processes (big-step time advance semantics).

Note that an ABS model that contains neither duration nor resource constructs will run without influencing the clock. This means that all ABS models are valid in Real-Time ABS; Real-Time ABS is a strict super-set of (untimed, non-resource-aware) Core ABS. ABS models without duration or resource constraints run in zero simulated time; the clock will be at time zero at the end of the simulation.

The standard library contains the following definition of a time datatype:

```
data Time = Time(Rat timeValue);

def Time now() = ...
```

The function **now**() always returns the current time as a non-negative rational number.

2.1 Basic Manipulation of Time in ABS

The following statements cause simulated time to advance:

```
duration(Rat min, Rat max);

await duration(Rat min, Rat max);
```

The **duration** statement blocks the cog of the executing process until at least min and at most max time units have passed. The **await duration** statement suspends the current process until at least min and at most max time units have passed.

In practice, the main purpose of these statements for directly controlling time advance is to model external processes that are not part of the model (e.g., a database query, machine initialization, ...). In all these cases, the exact duration might be known only within an interval, which is why these statements take two arguments. The language semantics and simulators guarantee that time will not advance beyond max, but not that it will advance precisely to the min of the interval.

The difference between **duration** and **await duration** is that in the latter case other processes in the same cog can execute while the awaiting process is suspended. In the case of the blocking **duration** statement, no other process in the same cog can execute. Note that processes in other cogs are not influenced by **duration** or **await duration**, except when they attempt to synchronize with the future of that process, at which point they will be blocked or suspended as well.

A subtle difference between **duration** and **await duration** is that in the latter case, the suspended process becomes eligible for scheduling after the specified time, but there is no guarantee that it will actually be scheduled at that point. This means that in case of multiple processes eligible for scheduling, a process might observe more time passed than the maximum given in its **await duration** guard.

Example:

```
Time t = now();
await duration(1/2, 5);
Time t2 = now();
```

In line 2, the process suspends for 1/2–5 time units; t2 will be at least 1/2 time units larger than t. The process will become enabled and runnable at most after 5 time units.

Example:

```
Time t = now();
duration(1/2, 5);
Time t2 = now();
```

In this example, the process and its cog will be blocked for 1/2–5 time units; t2 will be between 1/2 and 5 time units larger than t.

Chapter 3

Resource Modeling in ABS

Resource Modeling deals with simulating and analyzing the non-functional properties of models: planning and executing code deployment on varying numbers and kinds of machines, and the effects of different CPU speeds, interconnection bandwidth, code locality, etc. on the performance of a system. We call resources that influence deployment decisions *static* resources, and resources that influence performance *dynamic* resources. This section describes the constructs ABS offers to the modeler that form the base of qualitative and quantitative modeling of resources.

We introduce several tightly-connected concepts: *resource types* and resource annotations using them (Section 3.1), *resource configurations* (Section 3.2), and *deployment components* (Section 3.3). In brief, deployment components provide a resource configuration to cogs and their processes; resource configurations assign quantities to (a subset of all) resource types, and resource annotations describe semantic effects of parts of the model on a resource configuration (e.g., consuming some CPU by executing a certain statement).

All language identifiers described in this section reside in the ABS.DC module. To use them, import this module as follows:

```
module Name;
import * from ABS.DC;
```

3.1 Resource Types

The term *Resource* can be used in different ways. In ABS, we understand a Resource to be a countable, measurable property of a location. In the case of ABS, locations are modeled by deployment components. If the resource is influenced by program execution and the passage of time, we call it a *dynamic* resource. If the resource is unchanging and influences deployment, via its presence or quantity, we call it a *static* resource.

A resource always has a *resource type*; the simulation tools and language semantics currently offer bandwidth, CPU speed and memory as dynamic resource types, and memory and number of cores as static resource types. These resource types are defined in the ABS.DC module as follows:

```
data Resourcetype
= Speed
| Cores
| Bandwidth
| Memory
;
```

Notably, the Speed and Cores resource types started out as a unified resource type CPU. During work on modeling the Envisage case studies, it became apparent that there was a need to express both the number of cores of a VM and its relative speed, and that these concepts were only weakly related.

Note that there are some other attributes that a deployment component can carry: duration of starting up and shutting down, payment interval and cost per interval. These attributes model cost of deployment and are discussed in Deliverable D1.3.2.

Speed

The Speed dynamic resource type models execution speed. Intuitively, a deployment component with twice the number of CPU (speed) resources will execute twice as fast. In an ABS model, not all parts of a model consume CPU while executing – speed resources are consumed when execution in the current process reaches a statement that is annotated with a Cost annotation. This aligns with the observation that most CPU time will be spent at certain hotspots in a program; an example of this technique is presented in [4]. The advantage of explicit annotations is that, similar to running untimed models in Real-Time ABS, all un-annotated or partially annotated ABS models are valid and well-behaved under resource simulations.

Statements that consume CPU are annotated with a Cost annotation:

Example:

```
[Cost: 5] skip;
```

Executing the above **skip** statement will consume 5 Speed resources from the deployment component where the cog was deployed.

If the resource configuration of the deployment component does not have infinite speed resources, executing the above **skip** statement might take an observable amount of time proportional to its cost. For example, if the resource configuration contains 3 speed resources, the **skip** statement will be executed within $\lceil 5/3 \rceil = 2$ time units.

3.1.1 Bandwidth

Bandwidth is a measure of transmission speed. Bandwidth resources are consumed during method invocation and **return** statements. Bandwidth resources are consumed on both the sending and the receiving deployment component. We do not currently distinguish outgoing from incoming bandwidth since the case studies do not demand this distinction, so both incoming and outgoing messages consume from the same "budget".

Bandwidth consumption is expressed via a Size annotation to method invocation and return statements:

Example:

```
[Size: 2 * length(x)] o!m(x);

[Size: 1] return 145;
```

Executing the above method invocation statement will consume bandwidth resources proportional to the length of list x. The **return** statement consumes a constant amount of bandwidth during execution. Resource consumption will occur both at the sender (the deployment component where the current process is running), and at the receiver (the deployment component where o is deployed resp. where the return value is delivered).

Similar to the CPU speed case, executing the above statements will take an observable amount of time proportional to the message sizes given in the Size annotations. The effective bandwidth between two deployment components is the minimum of the bandwidths in the two resource configurations. Transferring a message will be finished when the necessary number of bandwidth resources has been consumed. If one deployment component has more bandwidth resources than the other, that deployment component can send and receive other messages concurrently since not all available bandwidth will be consumed.

3.1.2 Memory

The memory resource type abstracts from the size of main memory, as a measure of how many and which cogs can be created on a deployment component. The memory resource type is different from CPU and bandwidth in that it does not refresh in each time unit: memory that is consumed stays consumed until it is freed. Also in contrast to bandwidth and cpu, memory does not influence the timed behavior of the simulation of an ABS model. Instead, accidental or on-purpose overuse of memory in a model leads to failure. Hence, memory can be seen as both a dynamic and static resource type: it has an influence at runtime but does not influence timing behavior.

The necessary memory of running a new cog of a certain class (including the local objects and processes it creates) are specified with a MaxSize resource annotation.

Example:

```
[MaxSize: 5] class C { }
```

A new cog of the above (empty) class C can only be instantiated on a deployment component with at least 5 available memory resources.

3.1.3 Cores

The number of cores present on a deployment component is expressed via the Cores resource type. This static resource type is used in static and dynamic deployment to express and check constraints regarding number of cores needed for deployment of a module or system. This resource has influence on the simulation only insofar as it might influence deployment decisions (and hence the number of deployment components created). For static and dynamic deployment decisions, please refer to Deliverable D1.3.2.

3.2 Resource Configurations

To express availability of resource types, we introduce the concept of *resource configuration*. A resource configuration is a mapping from resource type to a number, for example map[Pair(Speed, 10), Pair(Bandwidth, 20)]. We use the standard ABS Map datatype; the ABS type of a resource configuration is Map<Resourcetype, Rat>, i.e., a mapping from resource types to rational numbers. Resource types not included, such as Memory in the above example, are treated as being infinite.

Example:

```
def Map<Resourcetype, Rat> amazonSmallInstance() =
  map[Pair(Speed, 100), Pair(Memory, 1000), Pair(Cores 2)];
```

In this example, we define a resource configuration as the result of the constant function amazonSmallInstance. This resource configuration models 100 Speed, 1000 Memory and (implicitly) infinite Bandwidth resources on a 2-core (virtual or physical) machine.

3.3 Deployment Components

Modeling code deployment and code execution under resource constraints requires a notion of *locality*. For this purpose, ABS offers a language construct called Deployment Component.

Deployment Components are first-class language constructs, i.e., they can be created, referenced and interacted with from within the model. A reference to a deployment component is treated the same way as a reference to an object. Deployment Components are created using the **new** expression. Any other cog can be created "on" a deployment component by using a DC annotation to the **new** statement.

A new deployment component is constructed using a name and a resource configuration.

Example:

```
1 DeploymentComponent dc = new DeploymentComponent("Server 1", amazonSmallInstance());
2 [DC: dc] Worker w = new CWorker();
```

Line 1 A new deployment component dc is created using the resource configuration defined earlier

Line 2 w will run inside dc; resource annotations inside the CWorker class will influence dc

Note that it is an error to try to locally create deployment components (via **new** local DeploymentComponent(...)) or new local objects on another cog (via [DC: x] **new** local C();).

3.4 Modeling Resource Usage

As described above, resource models are added to an ABS model using annotations. Adding annotations to specific statements and declarations causes side-effects on the status of an applicable deployment component.

Example:

```
 1 module Test;
 2 import * from ABS.DC;
 3 interface I {
 4   Unit process();
 5 }
 6 [MaxSize: 3]
 7 class C implements I {
 8   Unit process() {
 9     [Cost: 10] skip;
10 }
11
12 {
13   DeploymentComponent dc = new DeploymentComponent("Server",
14     map[Pair(Speed, 5), Pair(Bandwidth, 10), Pair(Memory, 5)]);
15   [DC: dc] I i = new C();
16   [Size: 5] i!process();
17 }
```

Line 2 Make all necessary identifiers accessible in the current module

Line 6 Declare the memory needed to instantiate a cog of class C

Line 9 Executing this statement costs 10 CPU units; the execution time will depend on the resource configuration of the deployment component, and on other cogs executing in parallel on the same deployment component. With the resource configuration in this example, executing the **skip** statement will take Cost:10 / CPU:5 = two time units.

Line 15 Creating a new cog succeeds since the available memory (5) is more than the necessary memory (3). Trying to create a second cog of the same class would fail in the given resource configuration.

Line 16 Executing this method call consumes 5 Bandwidth resources. Since dc has 10 bandwidth per time unit and the main block operates in an unconstrained resource configuration, the message will be transmitted in the same time unit.

Chapter 4

Static Analysis for Resource Modeling

The simulation tools developed in the Envisage project already handle resource annotations and deployment components and their resource configurations. I.e., information on non-functional properties can be obtained by running ABS models inside the simulator. This chapter presents further connections between the presented resource models and the static analysis tools developed in the Envisage project.

4.1 Worst-Case Size Analysis for Bandwidth Annotations

The *transmission data sizes* analysis presented in [3] statically infers an upper bound on the amount of data that the different objects in a distributed system may transmit. It is integrated into the resource analyzer SACO [1] presented in the deliverable D.3.3.1 of the Envisage project. This analysis focuses on method invocations and return statements because only at those points there will be data transmission, so its results can be used to infer the Size annotations related to method invocations that express bandwidth consumption. In order to obtain these upper bounds, the analysis over-approximates the sizes of the data at the program points where methods are invoked and where the results are received, and then over-approximates the total number of messages. The final result obtained by the analysis contains the data transmitted between every pair of objects, but they are indexed by origin object, destination object and method invoked; so the annotations can be easily extracted. For example, consider the following program:

Example:

```
 1  module Test_Bandwidth;
 2  import * from ABS.DC;
 3  interface I {
 4     List<Int> filter(List<Int> list);
 5  }
 6
 7  class C implements I {
 8     List<Int> filter(List<Int> list) {
 9        List<Int> result = Nil;
10        while (list != Nil) {
11           Int h = head(list);
12           if ( * ) { result = Cons(h,result); }
13           list = tail(list);
14        }
15        return result;
16     }
17  }
18
19  { //main
20     DeploymentComponent dc = new DeploymentComponent("Server", map[Pair(Bandwidth, 10)]);
21     [DC: dc] I i = new C();
22     List<Int> list = Cons(1,Nil);
23     i!filter(list);
24  }
```

The developer would want to infer automatically the annotation of the invocation in Line 23. In this case the transmission data sizes included in SACO would produce the following results—they have been pretty-printed for clarity:

```
[main -> i, C.filter] = 3 // <1>
[i -> main, C.filter] = 3 // <2>
```

The line in the output marked <1> shows that there is a transmission of 3 units of data from the main block to the object i—note that the list Cons(1,Nil) has size 3: 2 list constructors plus one basic integer value—corresponding to the invocation at Line 23. In the worst case the list result that is returned at Line 15 will be as long as the list list passed as argument. Therefore the analysis expresses in the output line marked <2> that there is a transmission of 3 units of data from the object i to the main block.

Using this information the developer can add a Size annotation in the method invocation at Line 23:

Example:

```
[Size: 3] i!process(list);
```

Similarly with the **return** statement at Line 15:

Example:

```
[Size: 3] return result;
```

4.2 Worst-Case Analysis of CPU Resource Consumption

As not all statements consume CPU resources, the Cost annotations presented in Section 3.1 are used to indicate how many CPU resources are consumed when execution in the current process reaches a particular statement. Therefore it would be very interesting to obtain an upper bound on the total consumption of CPU resources in a method. The resource analysis presented in [2] and integrated into SACO [1] (deliverable D.3.3.1) can solve this problem. This analysis performs two steps: first it generates a set of cost relations from the program source code and then it solves them to obtain an upper bound on the resource consumption. A very interesting feature of the first step is that it is *parametric* on the notion of cost: it supports different metrics, known as *cost models*, in order to quantify the cost of an execution step. Thus it can measure different kinds of resources: the number of executed instructions, the number of objects created, the number of methods invoked, etc. In this case, the resource analysis could be easily extended with a new cost model that takes into account only annotated statements. This new cost model would quantify as 0 the cost of a statement without annotation, whereas a statement with annotation [Cost: c] would be quantified as c. Using this new cost model the resource analysis could generate cost relations containing the information from the annotations and then compute an upper bound on the total consumption of CPU resources without further changes.

Consider the following method:

Example:

```
Unit process(Int n) {
  while (n > 0 ) {
    [Cost: 7] skip; //Consumes 7 CPU resources
    n = n − 1;
  }
}
```

This method executes a statement of cost 7 which is inside a loop that iterates nat(n) times—where nat(n) = max(n, 0) is used to lift negative values to 0. When computing the resource analysis for the process method the result would be $7 * nat(n)$ CPU resources, as it was expected.

Example:

```
UB for 'process'(this,n) = 7*nat(n)
```

4.3 Upper Bounds and Simulation

The static analyses integrated in the resource analyzer SACO [1] infer upper bounds on the resource consumption of the different methods, using different cost models such as number executed instructions, size of transmitted data, number of objects created, number of methods invoked, etc. These upper bounds are usually complex (mathematical) cost expressions that involve the size of the method parameters, where the meaning of *size of a parameter* depends on the type of the parameter. Consider the following method powlist which takes a list l of integer values and an integer n, and returns a list whose elements are those of l raised to the power of n (using method pow):

Example:

```
List<Int> powlist(List<Int> l, Int n) {
    List<Int> result = Nil;
    while (list != Nil) {
      Int h = head(l);
      Fut<Int> f = this ! pow(h, n);
      Int pow_h = f.get;
      result = Cons(pow_h,result);
      l = tail(l);
    }
    return result;
  }

Int pow(Int b, Int n) {
    Int i = 0;
    Int acc = 1;
    while (i < n) {
      acc = acc * b;
      i = i + 1;
    }
    return acc;
  }
```

Analyzing the above code using SACO, we obtain the following upper bounds on, for example, the number of executed instructions:

Example:

```
UB for 'C.powlist'(this,l,n) = 6+nat(l)*(19+6*nat(n))
UB for 'C.pow'(this,b,n) = 6+6*nat(n)
```

As expected, the upper bound of method pow depends only on the parameter n, and the upper bound of method powlist depends on both l and n. Note that nat(v) should be interpreted as max(0,v). Variable l in the above cost expressions refers to the length of the list to which variable l is bounded, and variable n refers to the corresponding integer value (since the size of an integer is its value). Now suppose that we have concrete values for l and n, during simulation for example, and we would like to compute an upper bound on the cost of executing powlist(l, n). This can be done by first computing the length of the list l and then substituting the result, together with the value of n, in the above expressions.

Although computing upper bounds for concrete data is simple for the specific case above, for complex data-structures it might be more complicated as the definition of the corresponding size functions might be more elaborated. In order to simplify this process, and thus facilitate the usage of the inferred upper bounds during simulation, SACO generates ABS functions that can be used to compute corresponding upper bounds without the need of computing the corresponding sizes by the user (or the simulator). Each such function takes the same parameters as the corresponding method and returns the evaluation of the corresponding upper bound with respect to those parameters. For example, for the methods above SACO would generate the following code:

Example:

```
def Int nat(Int n) = max(0, n) ;

def Int listsize_list_int(List<Int> list) =
  case list {
     Cons(hd, tl) => 1 + listsize_list_int(tl);
     Nil => 0;
  } ;

def Int powlist_ub(List<Int> list, Int n) = 6 + nat(listsize_list_int(list)) ∗ (19 + 6 ∗ nat(n)) ;

def Int pow_ub(Int elem, Int n) = 6 + 6 ∗ nat(n) ;
```

Note that in addition to the functions that define the upper bounds for methods powlist and pow, namely powlist_ub and pow_ub, SACO generates a function listsize_list_int that computes the size of a list of type List<Int> (in this case its length). If we had more types in the program, corresponding size functions will be generated. These upper bound functions can be easily integrated into the simulation process by means of annotations. For example, consider the following main method that invokes powlist and note the Cost annotation at Line 3:

Example:

```
1 Unit main() {
2   List<Int> l = Cons(1, Cons(2, Nil));
3   [Cost: powlist_ub(l, 5)] this ! powlist(l, 5);
4 }
```

Now when reaching Line 3, if the simulator needs (an upper bound on) the cost of the corresponding call to powlist it can simply execute the call given in the corresponding Cost annotation, namely powlist_ub(l, 5).

4.4 Static Memory Analysis

The work included as Appendix B in Deliverable D2.2.1 and reported in [6] proposes a static analysis technique that computes upper bounds of resource consumption (the technique is tailored to virtual machine usage in order to measure the maximum number of virtual machines used by a cloud service that dynamically acquires and release these kind of resources; however the same technique can be applied to heap memory consumption). This technique is orthogonal to the MaxSize annotations presented in Section 3.1.2 but is a step towards checking the validity of such MaxSize annotations against the ABS class itself.

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.

The language features not only the allocation of memory (**new** operation) but also the deallocation (release operation). When deallocation is considered, the precision of the computed upper bounds is heavily affected by parallelism, which is intrinsic in ABS semantics (i.e. if several tasks are managing the memory allocation concurrently, the estimation of how many memory slots are free at any given point in the computation may require a considerable effort).

The technique in the paper can be adopted for generating the `Size` annotation for methods. Indeed, our behavioural types can be used for storing information on object creation and computing the maximum number of objects created within a cog in a certain class. This number can then be set as the `Size` annotation of the class.

In order to illustrate the features of our technique we discuss few examples. For every example we also examine the type of output we expect from our cost analysis. We present two methods computing the factorial function, declared in a (omitted) class Math:

```
[Size: n] Int fact(Int n){
    Fut<Int> x ; Int m ; Math z ;
    if (n==0) { return 1 ; }
    else { z = new Math() ; x = z!fact(n−1) ; m = x.get ; release z ; return n∗m ;
    }
}
[Size: 1] Int cheap_fact(Int n, Int r){
    Fut<Int> x, y ; Int m ; Math z ;
    if (n==0) { return r ; }
    else { z = new Math(); x = z!prod(n,r) ; m = x.get ; release z ;
            y = this!cheap_fact(n−1,m) ; await y? ; m = y.get ; return m ;
    }
}
```

(`prod(x,y)` has been omitted: it just returns `x*y`). The method `fact` is the standard definition of factorial with the recursive invocation `fact(n-1)` performed on a new object `z`. The caller waits for its result, let it be `m`, then it deallocate the object `z` and delivers the value `n*m`. Notice that every object creation occurs before any release operation. As a consequence, `fact` will create as many new objects (and new cogs) as the argument n. Therefore, in order to be executed, `fact` needs `n` additional memory slots, as it is displayed by the corresponding Size annotation.

While `cheap_fact` also computes the factorial, its behaviour is different. In particular it implements the function

$$f(n,r) \;=\; \begin{cases} r & \text{if } n = 0 \\ f(n-1, n*r) & \text{otherwise} \end{cases}$$

which is initially invoked with $f(n,1)$. In `cheap_fact` the computation of $n*r$ is performed on a new object, which is deallocated *before* the recursive invocation. For this reason, one might always *reuse* the same object (from a pool). In fact, it turns out that the cost of `cheap_fact` is `1`, as it is displayed by the Size annotation.

The following cost equations for our Math class are a simplified version of the automatic cost equation generation described in the Appendix B (The full detail of the analysis of these as well as of other more complex examples can be found in the demo website[1]):

```
eq(fact(N), 0, [N = 0]).
eq(fact(N), 1, [fact(N − 1)], [N > 0]).
eq(fact(N), 0, [],   [N > 0]).


eq(cheap_fact(N, R), 0, [N = 0]).
eq(cheap_fact(N, R), 1, [prod(N,R)], [N > 0]).
eq(cheap_fact(N, R), 0, [cheap_fact(N − 1)], [N > 0]).


eq(prod(N,R), 0, [], []).
```

The equations for both factorial functions are quite similar, the main difference is in number of objects created while the recursive invocations takes place. In the case of `fact` the recursive invocation (second equation above) takes place after the creation of one object (see value 1 in the second argument of the equation), this object is released just after the recursion concludes (see third equation). On the other hand in `cheap_fact` the recursive invocation (sixth equation)takes place after the release of the newly created object (see second argument of the equation).

The resolution of these equations by means of an automatic solver produce the expected results, exactly the values specified in the Size annotations.

[1]http://sra.cs.unibo.it

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. Cost Analysis of Concurrent OO programs. In Hongseok Yang, editor, *Proceedings 9th Asian Symposium on Programming Languages and Systems (APLAS 2011)*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, 2011.

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

[4] 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.

[5] Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. On the integration of automatic deployment into the ABS modeling language. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, volume 9306 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, 2015.

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

Glossary

Deployment Component A location where cogs and their processes execute. Deployment components have an associated resource configuration.

Dynamic Resource A resource that influences runtime performance.

Real-Time ABS ABS with a dense-time clock, `duration` guard and `await duration` statement.

Resource A property of a deployment component reflecting real-world machine capacities like CPU speed, bandwidth, available memory. Resources consist of a resource type and a positive number or infinity.

Resource Annotation An annotation in ABS code that expresses a resource need at the location of the annotation. The semantic effect is dependent on the resource type, but typically a number of available resources will be consumed upon execution.

Resource Configuration A mapping from resource type to rational number, designating available or needed amounts of resources. Resource types that are not contained in the resource configuration are treated as infinite.

Resource type A unit of resources, e.g., bandwidth or CPU usage.

Static Resource A resource that influences deployment decisions.