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

Deliverable D1.3.2
Modeling of Deployment (Final Report)

Date of document: T30

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

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

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 Deployment (Final Report)

This document summarizes deliverable D1.3.2 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This deliverable describes the final outcomes of Task T1.3, dedicated to the integration of deployment scenarios related to virtualization into the Abstract Behavioral Specification language ABS. The preliminary deliverable D1.3.1 at T18 already reported the main lines of research followed within this Task. Following the structure of the previous deliverable, besides an introductory chapter, this deliverable includes three main contributions organized in three independent chapters.

Chapter 2 focuses on an extension of the ABS language for programming deployment declaratively: the ABS language is enriched with annotations used by the programmer to specify high-level deployment constraints, and then an external solver synthesizes the ABS low-level deployment actions needed to realize a deployment satisfying the specified constraints. In our preliminary deliverable we already discussed this approach, but it was limited to the synthesis of the initial deployment of an ABS model. We have completed this line of work by supporting also dynamic modifications to the current deployment, thus managing the elastic up- and down-scale of the modeled application.

Chapter 3 reports a more mature experience, w.r.t. the one described in D1.3.1, about the use of ABS for the modeling and analysis of dynamic deployment strategies: we discuss the general outline of dynamic resource management in terms of using a given amount of resources and scaling to change the amount of resources. We also discuss the realization (and empirical validation) of ABS-YARN, i.e., a configurable ABS-based framework for the modeling of clusters using YARN, a state-of-the-art technology for job scheduling and resource management. In this case, the analysis is done by exploiting one of the ABS back-ends, namely, the MAUDE simulator and the empirical evaluation of the modeling framework by comparing benchmarks with a real cluster with 30 machines.

Finally, Chapter 4 reports on an updated version of the so-called *ABS Cloud API* (included in the ABS Standard Library) providing interfaces for dynamic acquisition/release of resources, as well as the dynamic inspection of the current state of resource usage. These interfaces have been used in the modeling of the case-studies, and then collected in an ABS library to help the ABS programmer in the modeling of issues related to deployment.

The deliverable includes also two technical appendixes, each one containing one paper. The first one, in Appendix A and currently submitted, is integral part of Chapter 2: it describes automatic declarative deployment for ABS, driven and validated by the FRH case study. The second one, in Appendix B and published in the Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE'16), describes the modeling of the YARN platform with ABS.

List of Authors

Rudolf Schlatte (UIO)
Einar Broch Johnsen (UIO)
Gianluigi Zavattaro (BOL)

Contents

Chapter 1

Introduction

Software applications deployed and executed on top of cloud computing infrastructures should flexibly adapt in order to dynamically acquire or release computing resources. This is necessary to properly deliver to the final users the expected services at the expected level of quality, maintenance an optimized usage of the computing resources (e.g., avoiding resource over-provisioning). For this reason, modern software systems call for novel engineering approaches that anticipate deployment issues already at the early stages of development.

Several modeling languages like TOSCA [15], CloudML [11], and CloudMF [10] have been proposed to specify the deployment of software artifacts, but they are mainly intended to express deployment of already developed software. An actual integration of deployment in software development is still far from being obtained in the current practices. To cover this gap, the Envisage project, in particular Task 1.3, addresses the problem of extending the ABS (Abstract Behavioural Specification) language with linguistic constructs and mechanisms to properly specify deployment. In the preliminary deliverable D1.3.1 we already outlined the approach adopted in ABS for the modeling of deployment:

> *Deployment components* are added to the ABS language to represent object containers, responsible for providing the running objects with the resources they require (e.g. CPU cores, memory, bandwidth, etc.). Primitives are also added, specified by means of the so-called *ABS Cloud API* (included in the ABS standard library), to deal with the management of deployment components, like e.g. their creation/destruction or the monitoring of the current resource usage.

More precisely, D1.3.1 contained three main contributions. The first one was concerned with the automatic static deployment of applications specified in ABS: starting from ABS classes annotated with deployment information (namely, the resources that objects of that class need to properly run), a description of the available deployment components indicating their resources and costs, and the high-level specification of the expected deployment, an external engine synthesizes ABS code (to be included in a main block) that generates an application instance satisfying the constraints by minimizing the total cost. The second contribution included the description of preliminary experiences in the usage of ABS to specify, analyse and compare different deployment strategies for cloud applications; in particular, ABS was used to model resource management policies, and then the ABS simulator allowed for the evaluation of the corresponding performances. The third contribution was a preliminary version of the *ABS Cloud API*.

Following the research directions reported in D1.3.1, this deliverable presents the completion of T1.3 activities by:

- discussing additional ABS extensions supporting the declarative programming of deployment, not only for the initial static application configuration, but also considering the so-called dynamic upscale and downscale of the application deployment;
- showing how ABS can be used to model and reason about deployment strategies on top of industrial frameworks, like Apache Hadoop YARN, representing state-of-the-art technologies for resource management and job scheduling;

- completing the specification of the *ABS Cloud API*.

In the remainder of this chapter we more precisely focus on the first two contributions (the *ABS Cloud API* does not require further introduction) and we outline the structure of the deliverable.

1.1 Declarative Elasticity in ABS

Following the general approach presented in [6], and initially applied to ABS in [7], we have adopted a declarative approach according to which the programmer specifies deployment constraints and a solver computes actual deployments satisfying such constraints. In particular, in our previous work [7] included in D1.3.1, we presented an external engine for ABS able to synthesize ABS code specifying the initial static deployment. In this deliverable we complete this work by fully integrating this approach in the ABS language allowing for the declarative specification of the dynamic upscale/downscale of the modeled application depending, e.g., on the monitored workload or the current level of resource usage.

Our initial proposal for the declarative modeling of deployment in ABS [7] was based on three main pillars: (i) classes are enriched with annotations that indicate functional dependencies of objects of those classes as well as the resources they require, (ii) a separate high-level language for the declarative specification of the deployment, (iii) an engine that, based on the annotations and the programmer's requirements, computes a fully specified deployment that minimizes the total cost of the system. The computed deployment is expressed in ABS and can be manually included in a main block.

The work in [7] had two main limitations: (i) there was no way to express dynamic deployment decisions like, e.g., the need to upscale or downscale the modeled system and (ii) there was no automatic integration of the code synthesized by the engine in the corresponding ABS specification. In this deliverable we address these limitations by promoting the notion of deployment as a first-class citizen of the language. During a pre-processing phase, a new tool named SmartDepl generates ad-hoc classes exposing the methods `deploy` and `undeploy` to dynamically upscale and downscale the system. The deployment requirements can now also reuse already deployed objects just specifying which of the existing objects could be used, and how they should be connected with new objects to be freshly deployed. This has been the fundamental step forward that allowed us to support dynamic modification of the current deployment. Moreover, other relevant novel contributions are: (i) a more natural high-level language for the specification of requirements that now supports universal and existential quantifiers, and (ii) the usage of the delta modules and the variability modeling features of the ABS framework [5] to automatically and safely inject the synthesized deployment instructions into the existing ABS code.

Our ABS extension and the realization of the corresponding SmartDepl tool have been driven and validated by considering the Fredhopper Cloud Services case study. Fredhopper Cloud Services offer search and targeting facilities on a large product database to e-Commerce companies. Depending on the specific profile of an e-Commerce company FRH has to decide the most appropriate customized deployment of the service. Currently, such decisions are taken manually by an operation team which decides customized, hopefully optimal, service configurations taking into account several aspects like, for instance, the level of replications of critical parts of the service to ensure high availability. The operators manually perform the operations to scale up or down the system and this usually causes the over-provision of resources for guaranteeing the proper management of requests during a usage peak. With our extension of ABS, we have been able to realize a new modeling of the Fredhopper Cloud Services in which both the initial deployment and the subsequent up- and down-scale is expected to be executed automatically. This new model is a first fundamental step towards a new more efficient and elastic deployment management of the Fredhopper Cloud Services.

1.2 Modeling and Analysis of Deployment Strategies in ABS

Shifting deployment decisions from the deployment phase to the design phase of a software development process calls for the possibility to perform model-based validation of the chosen decisions during the software design. However, virtualized computing poses new and interesting challenges for formal methods because we need to express deployment decisions in formal models of distributed software and analyze the non-functional consequences of these deployment decisions at the modeling level. A popular example of cloud infrastructure used in industry is Hadoop [2], an open-source software framework available in cloud environments from vendors such as Amazon, HP, IBM, Microsoft, and Rackspace. YARN [9] is the next generation of Hadoop with a state-of-the-art resource negotiator. We have developed ABS-YARN, a generic framework for modeling YARN infrastructure and job execution. Using ABS-YARN, modelers can easily prototype a YARN cluster and evaluate deployment decisions at the modeling level, including the size of clusters and the resource requirements for containers depending on the jobs to be executed and their arrival patterns. Using ABS-YARN, designers can focus on developing better software to exploit YARN in a cost-efficient way.

The basic approach to modeling resource management for cloud computing in ABS is a separation of concerns between the resource costs of the execution and the resource provisioning at (virtual) locations. Upon modeling of the resource management in ABS, we can use the executable semantics of ABS, defined in Maude, as a simulation tool. ABS-YARN has been defined to support easy-to-use rapid prototyping of YARN-based applications, in such a way that the ABS simulation tool can be subsequently used to perform evaluation of the designed application.

To validate ABS-YARN, we comprehensively compared the results of model-based analyses using our modeling framework with the performance of a real YARN cluster by using several Hadoop benchmarks to create a hybrid workload and designing two scenarios in which the job inter-arrival time of the workload follows a uniform distribution and an exponential distribution, respectively. The results demonstrate that ABS-YARN models the real YARN cluster accurately in the uniform scenario. In the exponential scenario, ABS-YARN performs less well but it still provides a good approximation of the real YARN cluster. The main contributions can be summarized as follows:

1. We introduce ABS-YARN, a generic framework for modeling software targeting YARN. Using ABS, designers can develop software for YARN on top of the ABS-YARN framework and evaluate the performance of the software model before the software is realized and deployed on a real YARN cluster.

2. ABS-YARN supports dynamic and realistic job modeling and simulation. Users can define the number of jobs, the number of the tasks per job, task cost, job inter-arrival patterns, cluster scale, cluster capacity, and the resource requirements for containers to rapidly evaluate deployment decisions with the minimum costs.

3. We comprehensively evaluate and validate ABS-YARN under several performance metrics. The results demonstrate that ABS-YARN provides a satisfiable modeling to reflect the behaviors of real YARN clusters.

1.3 Structure of the Deliverable

The remainder of this deliverable is composed of three sections, one for each of the three main contributions. Chapter 2 details SmartDepl, the new tool for the declarative programming of deployment in ABS, supporting also dynamic upscale/downscale. Chapter 3 presents ABS-YARN for the modeling of applications based on the Apache Hadoop YARN technology for resource management and job scheduling. Finally, Chapter 4 reports the description of the ABS Cloud API. The deliverable includes also two technical appendixes, each one containing a technical paper: the first one (in Appendix A) details SmartDepl, while the second one (in Appendix B) discusses ABS-YARN.

Chapter 2

Declarative Elasticity in ABS

In D1.3.1 we already discussed how to automatize the instantiation of the initial static deployment of an ABS model. The ABS code was synthesized by an external engine based on the Zephyrus tool [6]. Our main source of inspiration for that preliminary work has been provided us by the FRH case study, in particular the problem of customizing the deployment of instances of the Fredhopper Cloud Services based on the customer profile (e.g. the expected number of final clients, possible usage peaks, etc.). Reasoning about deployment at the modeling level can have several interesting benefits. For example, in the case of Fredhopper Cloud Services, it can be a valuable support to the decisions currently taken by the so-called operations team responsible to actually deploy the Fredhopper Cloud Services instances.

However, there are interesting aspects related to the FRH case study that we were unable to address in our preliminary work. In particular, the number of requests can vary greatly over time, and typically depends on several factors. Figure 2.1 is a typical real-world graph for a single day (with data up to 18:00) showing the number of queries per second (y-axis, ranging from 0-25 qps, the horizontal dotted lines are drawn at 5,10,15 and 20 qps) over the time of the day (x-axis, starting at midnight, the vertical dotted lines indicate multiples of 2 hours). A low in demand is clearly observable between 2am - 5am, and this typically occurs every day. Moreover, peaks can occur during promotions of the shop or around Christmas.

This dynamic variation of the number of requests over time in the FRH case study, suggested us to improve our previous work to support also automatic *dynamic deployment*, namely its upscale and downscale. This extension required a complete re-design of our approach. In our preliminary work, the external deployment engine synthesized ABS code to be manually copy-pasted in the ABS specification. This is not a viable solution for dynamic deployment modifications because, for instance, upscale actions executed within a loop should be distinct between one cycle and the subsequent one. For this reason, we have promoted the notion of deployment as a first-class citizen of the language. A deployment is now an object on which methods like `deploy`, or `undeploy`, can be invoked to execute, or cancel, the corresponding deployment actions. For instance, in case of a loop, a new distinct deployment object can be instantiated at each cycle.

The external engine generates the class declaration for the deployment objects. In the ABS specification, wherever deployment actions are needed, the programmer can declaratively specify a corresponding deployment class. More precisely, two kinds of information are expressed: (i) the constraints on the new target configuration and (ii) the objects and deployment components already available that can be used in the new configuration. The external engine generates the ABS code for the required deployment classes as follows: it first synthesizes the new configuration (minimizing the costs for new deployment components) and then generates an ABS class implementing a predefined SmartDeployInterface, that includes the above mentioned `deploy` and `undeploy` methods. Instead of using manual copy-paste on the generated code, we now use the delta modules and the variability modeling features of the ABS framework [5] to automatically and safely inject the deployment instructions into the ABS specification.

It is worth to mention that we also re-engineered the external deployment engine by realizing a new tool named SmartDepl. Such tool uses Zephyrus2 (freely available at `https://jacopomauro@bitbucket.org/jacopomauro/zephyrus2.git`) which is a new implementation of the Zephyrus tool that was used in

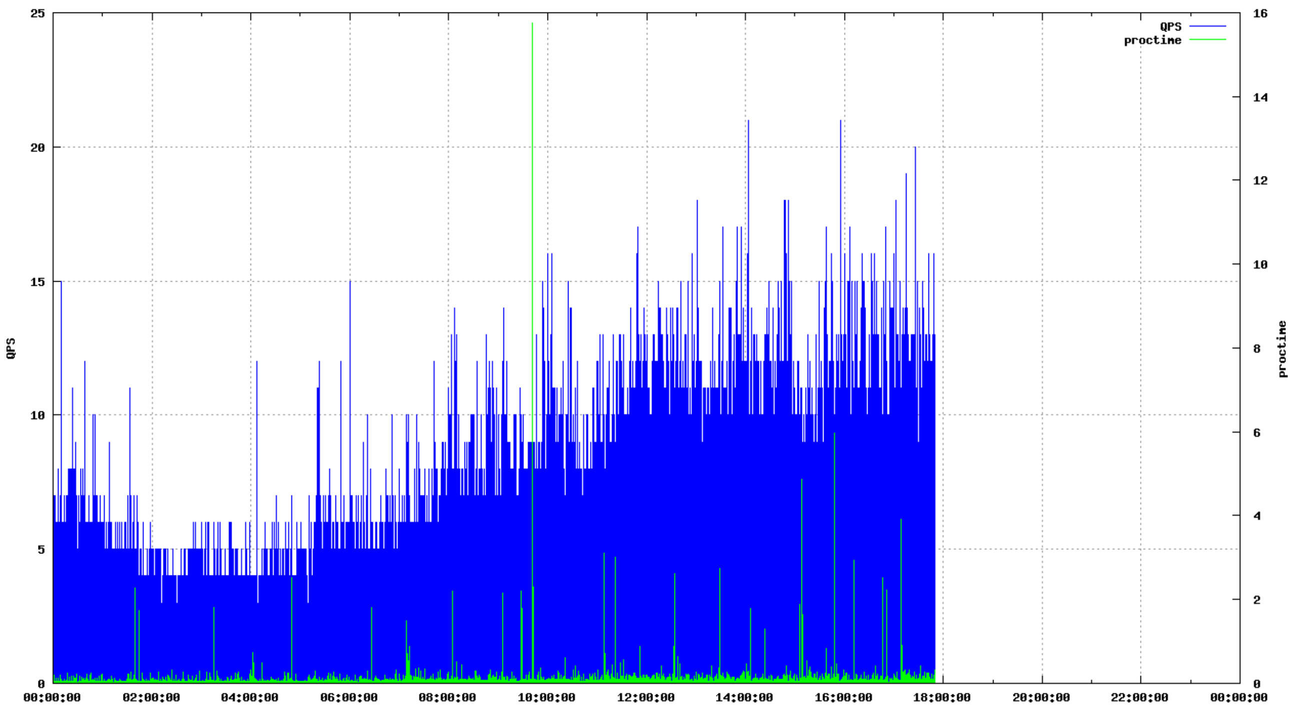


Figure 2.1: The number of queries per second over time (green part represents the query processing time).

our previous work. Zephyrus2 supports a more expressive language for declaration of the constraints on the desired configuration. In particular, the new language includes also universal and existential quantifiers. Another more technical novelty is concerned with the exploitation of MiniSearch [16] to support a more efficient and flexible framework for planning search strategies.

In the remainder of this Chapter we describe more precisely our contribution by (i) reporting the way functional dependencies and resource consumption annotations can be added to ABS classes, (ii) describing how to use and how we implemented the SmartDepl tool and (iii) showing its application to the modeling of dynamic deployment issues in the FRH use case.

2.1 Annotated ABS

The way deployment information are included in an ABS specification was not changed since the previous preliminary deliverable. Here, we quickly report a simple example of how deployment components are modeled according to the new version of the ABS Cloud API (described in Chapter 4) and how classes can be annotated with deployment information. For a detailed description we refer the interested reader to [7] (which was also a technical annex of D1.3.1).

As already mentioned, the basic element to capture deployment in ABS is the *deployment component*, which is a container for objects. A deployment component, intuitively, may model a virtual machine running objects that may represent the possible services that are offered by the virtual machine. The ABS Cloud API allows the programmer to model a cloud provider as a supplier offering a given set of deployment components, each one with its own resources and cost.

```
CloudProvider cProv = new CloudProvider("Amazon");
cProv.addInstanceDescription(Pair("c3",
  InsertAssoc(Pair(CostPerInterval,210),
    InsertAssoc(Pair(Memory,7500),
      InsertAssoc(Pair(Cores,4), EmptyMap)))));
DeploymentComponent dc = cProv.prelaunchInstanceNamed("c3");
[DC: dc] Service s = new QueryServiceImpl();
```

In the ABS code above, the cloud provider "Amazon" is modeled as the object `cProv` of type `CloudProvider`. The fact that "Amazon" can provide a virtual machine of type "c3" is captured by calling the `addInstanceDescription` in Line 2. With this instruction we also specify that c3 virtual machines cost 210 cents per hour and provide 7.5 GB of RAM and 4 cores. In Line 6 an instance of "c3" is launched and the corresponding deployment component is saved in the variable `dc`. Finally, in Line 6, a new object of type `QueryServiceImpl` (implementing the interface `Service`) is created and deployed within the deployment component `dc`.

In ABS it is possible to declare interface hierarchies and define classes implementing them.

```
interface Service { ... }
interface IQueryService extends Service { ... }
class QueryServiceImpl(DeploymentService ds, Bool staging)
  implements IQueryService { ... }
```

In the excerpt of ABS above, the `IQueryService` service is declared as an interface that extends `Service`, and the class `QueryServiceImpl` is defined as an implementation of this interface. Notice that the initialization parameters required at object instantiation are indicated as parameters in the corresponding class definition.

Classes can be enriched with annotations that denote the resources consumed and the functional requirements of an object of that class.

```
[ Deploy: scenario[Name("staging"), Cost("Cores", 2), Cost("Memory",7000),
  Param("staging", Default("True")), Param("ds", Req)] ]
[ Deploy: scenario[Name("live"), Cost("Cores", 1), Cost("Memory",3000),
  Param("staging", Default("False")), Param("ds", Req)] ]
```

The previous two annotations, assumed to be associated to the declaration of the class `QueryServiceImpl`, describe two possible deployment scenarios for the objects of that class. The first annotation captures the deployment of a Query Service in *staging* modality while the second one captures the deployment of the Query Service in *live* modality. A Query Service in staging modality requires 2 cores and 7GB of RAM while in live mode it only requires 1 core and 3GB of RAM. The creation of a Query Service Object requires an object of type `DeploymentService` that is associated with the parameter `dc` while the parameter `staging` is set to true or false according to the desired deployment scenario.

2.2 Deployment Declaration and Synthesis

When a system deployment should be automatically computed, a user must first specify the specific goals he expects to reach. For instance, in the considered Fredhopper Cloud Services use case, the initial goal is to deploy a given number of Query Services and a Platform Service, possibly located on different machines (e.g., to improve fault tolerance) and later on to upscale or downscale the system according to the monitored traffic.

All these goals can be expressed in the *Declarative Requirement Language* (DRL): a new language for stating the constraints that the final configuration should satisfy.

As shown in Table 2.1 that reports the DRL grammar defined using the ANTLR tool,[1] a goal is expressed as a boolean formula `b_expr` obtained using the usual logical connectives over comparison of arithmetic expressions. The atom of this arithmetic expression may be integers (Line 6), a quantifier statement (Line 7), a sum statement (Line 8) and an identifier for the number of deployed objects (Line 9). The number of objects deployed using a given scenario is defined by its class identifier and the scenario name enclosed in square brackets (Line 12).

As an example the following formula requires the presence of at least an object of class `QueryServiceImpl` deployed in staging mode.

```
QueryServiceImpl[staging] > 0
```

The square brackets may be omitted (Line 12 - first option) for objects that have only one default deployment scenario. Regular expressions (`RE` in Line 12) can be used to match objects deployed using

[1]ANTLR (ANother Tool for Language Recognition) - `http://www.antlr.org/`

```
1  b_expr : b_term (bool_binary_op b_term )* ;
2  b_term : ('not')? b_factor ;
3  b_factor : 'true' | 'false' | relation ;
4  relation : expr (comparison_op expr)? ;
5  expr : term (arith_binary_op term)* ;
6  term : INT                                        |
7    ('exists' | 'forall') VARIABLE 'in' type ':' b_expr |
8    'sum' VARIABLE 'in' type ':' expr                |
9    (( ID | VARIABLE | ID '[' INT ']' ) '.')? objId     |
10   arith_unary_op expr                              |
11   '(' b_expr ')'                                   ;
12 objId :  ID | VARIABLE | ID '[' INT ']' | ID '[' RE ']';
13 type : 'obj' | 'DC' | RE ;
14 bool_binary_op : 'and' | 'or' | 'impl' | 'iff' ;
15 arith_binary_op : '+' | '-' | '*' ;
16 arith_unary_op : 'abs' ; // absolute value
17 comparison_op : '<=' | '=' | '>=' | '<' | '>' | '!=' ;
```

Table 2.1: DRL grammar.

different deployment scenarios. The number of deployed objects can also be prefixed by a deployment component identifier to denote just the number of objects defined within this specific deployment component. As an example, the deployment of exactly one object of class `DeploymentServiceImpl` on the first and on the second instance of a "c3" virtual machine can be enforced as follows.

```
c3[0].DeploymentServiceImpl = 1 and
  c3[1].DeploymentServiceImpl = 1
```

Here the 0 and 1 numbers between the square brackets represent respectively the first and second virtual machine of type "c3" . To shorten the notation, the `[0]` can be omitted (Line 9).[2]

It is possible to use also quantifiers and sum expressions to capture more concisely some of the desired properties. Variables are identifiers prefixed with a question mark. As specified in Line 13, quantifiers and sum term variables can range on all the possible objects (`'obj'`), all the deployment components (`'DC'`), or just on all the virtual machines which names match a given regular expression (`RE`).

Using such constraint it is possible to express more elaborate constraints like the co-location or distribution of objects, or limit the amount of objects deployed on a given DC.[3] As an example, to enforce the constraint that every Query Service requires a Deployment Service installed on its virtual machine we can require the following.

```
forall ?x in DC: (
  ?x.QueryServiceImpl['.*'] > 0  impl
  ?x.DeploymentServiceImpl > 0
)
```

Here we use the regular expression `'.*'` to match with all the possible deployment modalities (either `staging` or `live`) for the Query Services. Note that the syntactic element `'impl'` denotes logical implication.

Finally, requiring for instance the load balancer to be installed alone in a virtual machine can be done as follows.

```
forall ?x in DC: (
  ?x.LoadBalancerServiceImpl > 0 impl
  (sum ?y in obj: ?x.?y) = ?x.LoadBalancerServiceImpl
)
```

[2]We assume that the user could launch only a bounded number of deployment components. In particular, for every cloud deployment type SmartDepl allows to specify the maximal number of deployment components that can be created.

[3]DRL improves on the specification language presented in [7] because the addition of the quantifiers and the sum terms allows the user to write her desiderata in a more concise and natural way.

```
 1 { "id": "AddQueryDeployer",
 2   "specification": "QueryServiceImpl[live] = 1",
 3   "obj": [ { "name": "platformObj",
 4              "provides": [ {
 5                "ports": [ "MonitorPlatformService",
 6                           "PlatformService" ],
 7                "num": -1 } ],
 8              "interface": "PlatformService" },
 9            { "name": "loadBalancerObj",
10              "provides": [ {
11                "ports": [ "LoadBalancerService" ],
12                "num": -1 } ],
13              "interface": "LoadBalancerService" },
14            { "name": "serviceProviderObj",
15              "provides": [ {
16                "ports": [ "ServiceProvider" ],
17                "num": -1 } ],
18              "interface": "ServiceProvider" } ],
19   "DC": [] }
```

Table 2.2: JSON annotation example.

SmartDepl is the tool that we have implemented to automatize deployment. More concretely, we require the program to specify all its deployment needs by enriching the ABS code with specific annotations (see Table 2.2 for an example of such annotations that we will describe later on). SmartDepl processes them and generates, for every deployment need, a new class that specifies the deployment steps to reach the desired target. This class can be used to trigger the execution of the deployment but also to undo it in case the system needs to downscale.

As an example, imagine that an initial deployment of the Fredhopper Cloud Services has been already obtained and that, for example due to a usage peak, it is necessary to add 1 Query Services in live mode. The annotation required by SmartDepl for capturing this need is the JSON object defined in Table 2.2.

In Line 1, by using the keyword `"id"` the programmer specifies that the name of the class containing the deployment code is `AddQueryDeployer`. As we will see later, the user can exploit this name to upscale and downscale the system assuming the class existence. The second line contains the desired configuration in DRL. By using the keyword `"obj"`, Lines 3-18 define objects that are assumed to be already available, hence are not needed to be re-deployed. Assuming that the user has already a working Fredhopper Cloud Services, he knows indeed that there is already a Platform Service, a Load Balancer and a Service Provider deployed. Every available object is defined by assigning to it a unique name (keyword `"name"` in Lines 3,9,14), the interfaces it provide (keyword `"port"` in Lines 5-6,11,16) with the amount of other objects that can use them (keyword `"num"` in Lines 7,12,17 — in this case a -1 value means that the object can be used by an unbounded number of other objects), and the object type (keyword `"interface"` in Lines 8,13,18). Finally, by using the keyword `"DC"` it is possible to specify if there are existing deployment components with free resources that can be used to deploy new objects inside them. In this case, for fault tolerance reasons we want to deploy the new objects in new machines and therefore `"DC"` is left empty (Line 19).

For the interested reader, the formal specification of the JSON annotation is defined in `https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/spec/smart_deploy_annotation_schema.json`.

Once the annotation is given the programmer may freely use this class. For instance, the ABS code below upscales and downscales the system based on a monitor decision.

```
while ( ... ) {
 if ( monitor.scaleUp() ) {
    SmartDeployInterface depObj = new AddQueryDeployer(cProv, platformService, loadBalancerService, serviceProvider);
    depObj.deploy();
    depObjList = Cons(depObj,depObjList);
  } else if ( (monitor.scaleDown()) && (depObjList != Nil) ) {
```

```
7       SmartDeployInterface depObj = head(depObjList);
8       depObjList = tail(depObjList);
9       depObj.undeploy(); } }
```

The idea is to store the references to deployment decisions in a list called depObjList. When the monitor decides to upscale by adding new Query Services (Line 2) a new deployment object is created (Line 3). In this case AddQueryDeployer is the name associated with the annotation previously discussed. Its first parameter is the cloud provider, as defined for instance in Section 2.1. The following parameters are the objects already available for the deployment that do not need to be re-deployed from scratch. These are given according to the order they are defined in the annotation in Table 2.2. The interface of this class is SmartDeployInterface which is initially an empty interface that SmartDepl populated with: i) a deploy method to realise the deployment of the desired configuration, ii) an undeploy method to undo the deployment gracefully by removing the virtual machine created with the application of the deploy method, iii) getter methods to retrieve the new list of objects and deployment components created by running the deploy method (e.g., to retrieve the list of all the Query Services created by depObj.deploy() it is possible to call the operation depObj.getIQueryService()). The real addition of the Query Service is performed in Line 4 with the call of the deploy method. If instead the monitor decides to downscale (Line 6), the last deployment solution is retrieved (Line 7) and then the action performed by the deployment are undone by calling the undeploy method.[4]

Technically, SmartDepl is written in python (∼1k lines of code) and relies on Zephyrus2, a configuration optimizer that given the user desiderata and a universe of components was able to compute the optimal configuration satisfying the user needs.[5] SmartDepl uses the cost annotations as defined in Section 2.1 to compute a configuration that satisfies the user requirements minimizing the cost of the deployment components that need to be created and, in case of ties, minimizing also the number of created objects. Once a configuration is obtained, SmartDepl uses a topological sort to take into account all the object dependencies and establishes the correct sequence of deployment instructions to realise the computed configuration. SmartDepl generates the code of the classes and the methods to inject to the interface exploiting Delta Model techniques [5]. SmartDepl notifies the user in case no configuration can satisfy the desiderata, e.g., when the specification is too restrictive. Moreover, SmartDepl also notifies the user when it is unable to generate a sequence of deployment actions due to mutual dependencies between the objects.[6]

As an example the deploy code generated by SmartDepl for the annotation defined in Table 2.2 is the following.

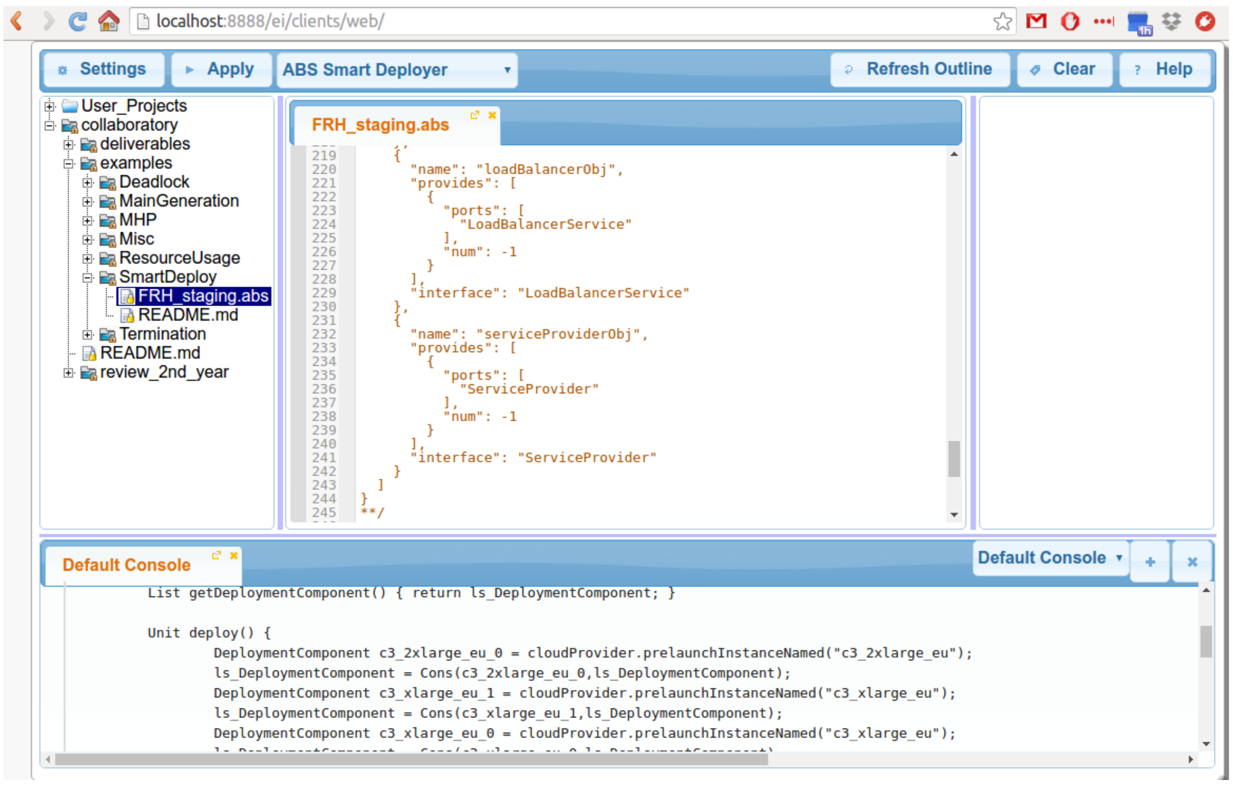
```
1  Unit deploy() {
2    DeploymentComponent c3_0 = cloudProvider.prelaunchInstanceNamed("c3");
3    ls_DeploymentComponent = Cons(c3_0,ls_DeploymentComponent);
4    [DC: c3_0] DeploymentService oDef___DeploymentServiceImpl_0_c3_0 =
5      new DeploymentServiceImpl(platformObj);
6    ls_DeploymentService = Cons(oDef___DeploymentServiceImpl_0_c3_0,
7      ls_DeploymentService);
8    [DC: c3_0] IQueryService olive___QueryServiceImpl_0_c3_0 = new
9      QueryServiceImpl(oDef___DeploymentServiceImpl_0_c3_0, False);
10   ls_IQueryService = Cons(olive___QueryServiceImpl_0_c3_0, ls_IQueryService);
11   ls_Service = Cons(olive___QueryServiceImpl_0_c3_0, ls_Service);
12   ls_EndPoint = Cons(olive___QueryServiceImpl_0_c3_0, ls_EndPoint);
13 }
```

In the previous code, at Line 3, a new deployment component c3_0 is created. At Lines 4-5 an object of class DeploymentService is created. This is due to the fact that every Query Service requires its Deployment Service (i.e., it is one of the required parameters, cfr. Section 2.1) and therefore this object needs to be created and

[4]Since ABS does not have an explicit operation to force the removal of objects the undeploy procedure just removes the references to these objects leaving the garbage collector to actually remove them. The deployment components created by the deploy methods are removed instead using an explicit kill primitive provided by ABS.

[5]SmartDepl Zephyrus2 (freely available at `https://jacopomauro@bitbucket.org/jacopomauro/zephyrus2.git`) is a completely new re-engineering of the previously used Zephyrus solver [6].

[6]This occurs when the creation of an object requires the execution of a complex protocol like for instance what happens for the boostrapping of Linux distribution [1].



```
    {
      "name": "loadBalancerObj",
      "provides": [
        {
          "ports": [
            "LoadBalancerService"
          ],
          "num": -1
        }
      ],
      "interface": "LoadBalancerService"
    },
    {
      "name": "serviceProviderObj",
      "provides": [
        {
          "ports": [
            "ServiceProvider"
          ],
          "num": -1
        }
      ],
      "interface": "ServiceProvider"
    }
  ]
}
**/
```

```
	List getDeploymentComponent() { return ls_DeploymentComponent; }

	Unit deploy() {
		DeploymentComponent c3_2xlarge_eu_0 = cloudProvider.prelaunchInstanceNamed("c3_2xlarge_eu");
		ls_DeploymentComponent = Cons(c3_2xlarge_eu_0,ls_DeploymentComponent);
		DeploymentComponent c3_xlarge_eu_1 = cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
		ls_DeploymentComponent = Cons(c3_xlarge_eu_1,ls_DeploymentComponent);
		DeploymentComponent c3_xlarge_eu_0 = cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
```

Figure 2.2: SmartDepl execution within the ABS toolchain IDE.

deployed before the Query Service. In Lines 8-9 the desired object of class IQueryService is finally created. Both tobjects are deployed on `c3_0`.

SmartDepl is open source and freely available from `https://github.com/jacopoMauro/abs_deployer/tree/smart_deployer`. As shown in Figure 2.2, SmartDepl has also been integrated into the ABS toolchain,[7] i.e., an IDE and a collection of tools for writing, inspecting, checking, and analyzing ABS programs developed withing the Envisage European project.

2.3 Application to the FRH use case

In this section we report about the modeling with SmartDepl of the concrete deployment requirements of the Fredhopper Cloud Services.

SmartDepl was used twice: to synthesize the initial static deployment of the entire framework and for dynamically adding (and then removing) single instances of Query Services in case the system needs to scale (up or down). Since Fredhopper Cloud Services is using Amazon EC2 Instance Types we used two types of deployment components corresponding to the "xlarge" and "2xlarge" instances of the Compute Optimized instances (version 3)[8] of Amazon. Moreover, for fault tolerance and stability, Fredhopper Cloud Services uses instances in multiple regions in Amazon (regions are geographically separate, so even if there is a *force majeure* in one region, other regions are not necessarily affected). We model the instance types in different regions as follows: "c3_xlarge_eu", "c3_xlarge_us", "c3_2xlarge_eu", "c3_2xlarge_us" ("eu" refers to a European region, "us" refers to an American region).

The static deployment of the system requires the deployment of a Load Balancer, a Platform Service, a Service Provider, 2 Query Services among whom at least one in staging mode. This can be easily expressed as follows.

[7]`http://abs-models.org/installation/`
[8]`https://aws.amazon.com/ec2/instance-types/`

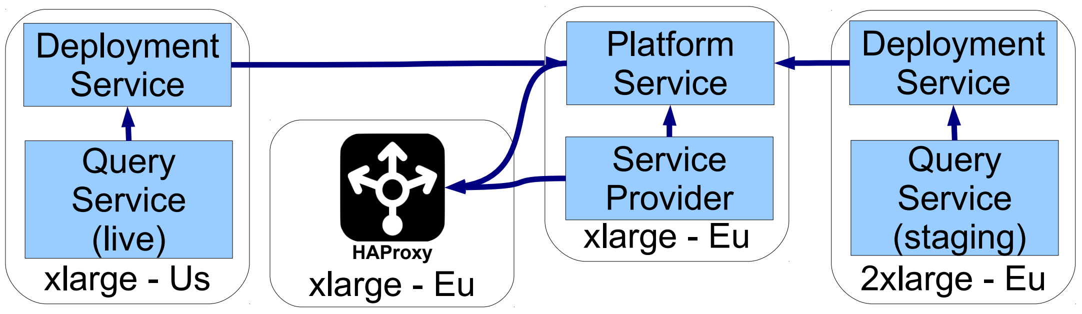


Figure 2.3: Example of automatic objects allocation to deployment components.

```
LoadBalancerServiceImpl = 1 and PlatformServiceImpl = 1 and
ServiceProviderImpl = 1 and QueryServiceImpl[staging] > 0 and
QueryServiceImpl[staging] + QueryServiceImpl[live] = 2
```

For the correct functioning of the system a Query Service requires a Deployment Service installed on the same machine. This constraint can be expressed as shown in Section 2.2. The requirement that a ServiceProvider is present on every machine containing a Platform Service can be expressed as follows.

```
forall ?x in DC: (?x.PlatformServiceImpl > 0 impl ?x.ServiceProviderImpl > 0)
```

Not all services can be freely installed on an arbitrary virtual machine. To increase fault tolerance Fredhopper Cloud Services require that the Load Balancer, the Query/Deployment Services, and the Platform Service/Service Provider are never co-located on the same virtual machine. This can be easily expressed as shown at the end of Section 2.2.

As mentioned above, to cope with catastrophic failures, Fredhopper Cloud Services distribute the Query Services among the available regions. This can be enforced by constraining the number of the Query Services in the different data centers to be equal. In DRL this can be expressed using regular expressions as follows.

```
(sum ?x in '.*_eu': ?x.QueryServiceImpl['.*']) =
(sum ?x in '.*_us': ?x.QueryServiceImpl['.*'])
```

Another constraint in the Fredhopper Cloud Services is that, for performance reasons, the Query Service in Staging mode should be located in the same region as that with the Platform Service, since Amazon connects instances in the same region with low-latency links. For the European data-center this can be expressed by:

```
(sum ?x in '.*_eu': ?x.QueryServiceImpl[staging]) > 0) impl
(sum ?x in '.*_eu': ?x.PlatformServiceImpl ) > 0)
```

With this specification, SmartDepl is able to compute that the initial configuration that minimizes the total costs per interval is the one depicted in Figure 2.3 that uses two "xlarge" instances in Europe for deploying the Load Balancer and the Platform Service, one "2xlarge" instance in Europe to deploy the Query Service in staging mode, and one "xlarge" instance in Europe to deploy the Query Service in live mode.

After this initial deployment, if there is need to scale up, two Query Service instances are added (one in an EU region, and one in an US region for balancing across regions). On the other hand, if there is unnecessary overcapacity, the most recent ones can be shut down. However, since the decision to scale is a manual process by the Cloud operations team, and FRH has very aggressive SLAs, the operations team is typically conservative with downscaling, leading to potential over-spending. The ability of SmartDepl to deploy in the programming language (ABS) itself allows to put auto-scaling on a rigorous basis. Furthermore, while the operations team currently use ad-hoc scripts to configure newly added or removed service instances, and these scripts are specific to the infrastructure provider, SmartDepl automatically generates code that accomplishes this (for example, see Table 2.2).

To automatically generate the desired deployment configuration, SmartDepl uses as specification all the previous constraints except that now instead of requiring a Platform Service and a Load Balancer we simply require two Query services in live mode. In this case, as expected after the deployment of the initial framework, the best solution is to deploy one Query Service in Europe and one in US using "xlarge" instances.

SmartDepl is able to compute the optimal deployment configurations and generate the code in less than 5 seconds. The ABS model used with all the annotations and specifications is available at `https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/test`

Chapter 3

Modeling YARN in ABS

3.1 Dynamic Resource Management

A common strategy for web applications these days, especially in early development and deployment, is to acquire the needed resources (server, storage, bandwidth) from a cloud infrastructure provider such as Amazon, Windows or Google, instead of purchasing server hardware and data centre space. In that way, initial costs can be kept low while still keeping the flexibility to react quickly to demand growth [4].

Deliverable D1.3.1 presented various examples of how resource management can be integrated in ABS models of resource-aware applications. In those examples we integrated the resource management strategies in the client layer (see Figure 3.1 which is taken from the DoW). The examples used a simple and initial version of a Cloud API which were only focused on computer resources and which was originally developed in [12, 13]. Chapter 4 discusses the current status of the Cloud API. In this chapter, we will now discuss in a high-level manner some dimensions of dynamic resource management as encountered during our work in the Envisage project.

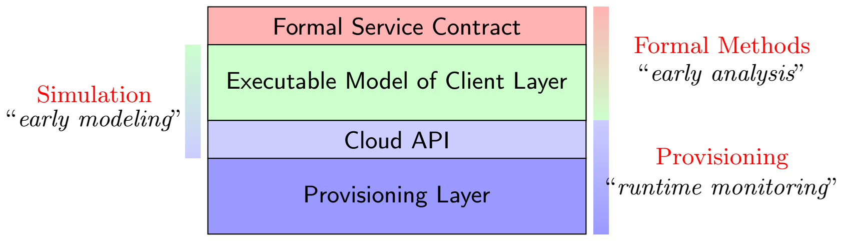


Figure 3.1: The approach to modeling services in Envisage.

3.2 Dimension 1: The Management of Available Resources

Dynamic resource management assumes that there is a number of resources available to the program. In the simplest case, this may simply be a single resource (e.g., a server), in which case the role of the dynamic resource manager would be to prioritise on the ordering of jobs sent to the server. Figure 3.2 depicts a typical scaling points for web applications: the workers that deal with the actual transactions from clients. These typically receive requests from a service endpoint which distributes html-queries to the workers. Note that there may be several other scaling points in an application. In the rest of this chapter, we shall focus on this scaling point and ignore other possible scaling points in the application.

A *resource manager* implements a policy for the utilisation of a pool of resources. There can be many different policies for resource management, both with respect to the selection of the job (which we call the *application management*) and the selection of the resource (which we call the *resource management*). For simplicity, we here assume that all jobs have equal priority and focus on the selection of the resource. (ABS

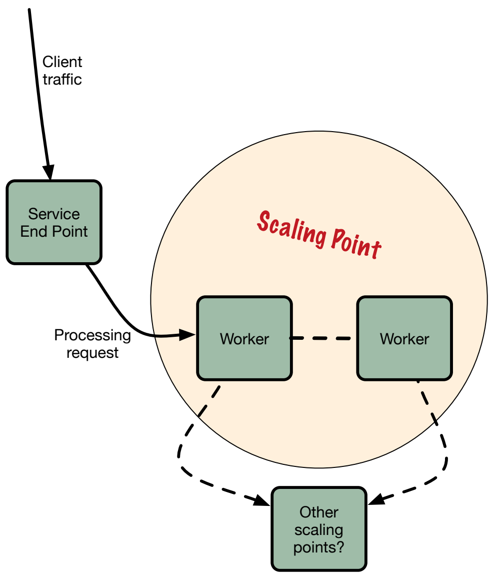


Figure 3.2: Scaling points.

supports customised schedulers for COGs which allow, e.g., priorities between jobs or earliest-deadline-first policies to be expressed at the abstraction level of the functional layer of the language [3].) For the selection of resources, we believe there are in practice basically two general strategies:

- *Round-robin*: the goal is to distribute jobs equally between the available resources
- *Saturation*: the goal is to fill the first resource up to capacity (e.g., a given level of acceptable load) before starting to use the next resource.

We have also experimented with more esoteric (i.e., highly application-specific) policies as one would get if different kinds of jobs have very different deadlines or the resources differ significantly in profile (e.g., a cheap, slow-running machine and an expensive, fast-running machine).

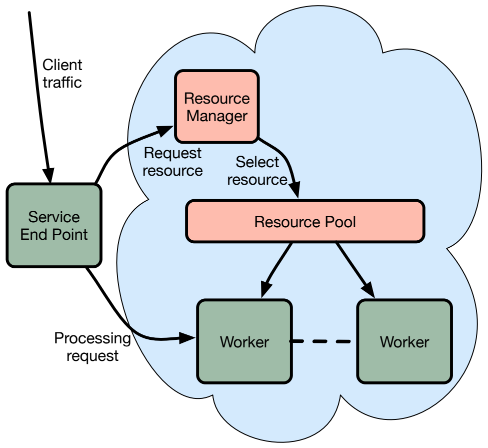


Figure 3.3: The resource manager.

A service running with dynamic resource management will typically have one COG implementing the application manager with the business logic and another COG implementing the resource manager, as depicted in Figure 3.3. This is not necessary but has the advantage of providing a separation of concerns such that the task selection as well as the resource manager can be replaced by another, implementing a different policy, without interfering with the rest of the system.

Many cloud systems restrict their dynamic resource management to this dimension of load distribution and fix the number of available resources before starting the application. One example of such a system is HADOOP, which in its modern version uses the YARN resource negotiator to manage the utilisation of resources.

We can illustrate a load balancer in ABS, realising the resource manager component of Figure 3.3, by the very simple class in Figure 3.4. The class implements the interface LoadBalancer with two methods for acquiring and releasing a resource, respectively. The application manager will interact with the resource manager by means of these methods. Here the Worker objects represent the application-level resources; i.e., the workers are deployed on one deployment component each, but this is transparent to the resource management policy. The class RoundRobinLoadBalancer implements a round robin resource management policy where a worker is never allocated for two requests at the same time. Note that a request for a worker will be suspended if the resource manager has no available workers.

Example:
```
interface LoadBalancer {
   Worker getWorker();
   Unit releaseWorker(Worker w);
}

class RoundRobinLB(List<Worker> resources) implements LoadBalancer {
    List<Worker> available = resources;

    Worker getWorker(){
        Worker w;
        await (available != Nil);
        w = head(available);
        available = tail(available);
        return w;
    }

    Unit releaseWorker(Worker w){
        available = appendright(available,w);
    }
}
```

Figure 3.4: A class implementing a simple resource manager in ABS

3.3 Dimension 2: Scaling

Scaling is a dimension of dynamic resource management which is orthogonal to the previous discussion. Scaling is concerned with fixing the number of resources available to the resource manager. Scaling may be a static decision, a manual decision at runtime, or the decision-making may be integrated in the application (so-called auto-scaling). For example in HADOOP YARN, the number of slave nodes for the big data processing is fixed in advance. For the Fredhopper case study of Envisage, the current production system does manual scaling. Auto-scaling requires a good understanding of thresholds for congestion and for the increase in expenses, and we believe a model-based approach such as the one developed in Envisage may help in gaining confidence in the adequacy of a considered auto-scaling algorithm.

The auto-scaling component depicted in Figure 3.5 is primarily responsible for adding or removing resources from the pool managed by the resource manager. Before we illustrate how an auto-scaling component may be realised in ABS, we observe that since the resource manager encapsulates the queue of available resources, the actual addition or removal of resources in our example must be handled by RoundRobinLoadBalancer (or the queue must be made external to the two components). Hence, we first

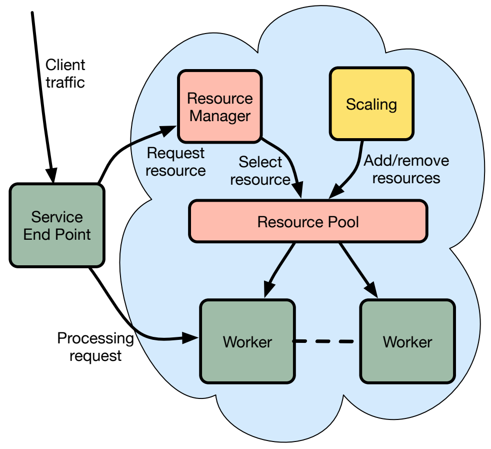


Figure 3.5: Scaling complements the resource manager.

revise our class from Figure 3.4 with a few additional features. The revised class is depicted in Figure 3.6 on Page 20. The new interface ResourceController exports methods to add or remove a worker with an associated deployment component). The class RoundRobinLB is now interacting with the Cloud API via the class parameter cloud to acquire and release virtual machine instances with a given resource specification spec. In this example, we also let the RoundRobinLB report to the auto-scaler component at regular intervals about the length of its queue of available workers. This is done by the run method.

We can now illustrate how a scaling component may be implemented in ABS by the class ScalingManager below. In this example, the scaling component receives updates about the length of two queues: one for the number of pending tasks, as monitored by the application manager, and the other for the number of available resources as discussed above. The auto-scaling strategy decides to add or remove deployment components depending on thresholds *upper* and *lower* for the ratio between the two queues.

Example:

```
interface ScalingAPI {
    Unit jobQueueLength(Int n);
    Unit workerQueueLength(Int n);
    Unit register(WorkManager rm);
}
class ScalingManager(Rat lower, Rat upper) implements ScalingAPI {
    Int jobQueue = 0;
    Int workerQueue = 0;
    ResourceManager resourcemanager;

    Unit run(){
        await duration(1,1); // activate at certain intervals (here every time interval)
        // check conditions for scaling up
        if (jobQueue > upper * workerQueue) { resourcemanager!addWorker(); }
        // check conditions for scaling down
        if (jobQueue < lower * workerQueue) { resourcemanager!removeWorker(); }
        // wait for time to pass, then repeat
        this!run();
    }

    Unit register(ResourceManager r){resourcemanager = r;}
    Unit jobQueueLength(Int n){jobQueue=n;}
    Unit workerQueueLength(Int n){workerQueue=n;}
}
```

Example:

```
interface LoadBalancer {
   Worker getWorker();
   Unit releaseWorker(Worker w);
}
interface ResourceController {
    Unit addWorker();
    Unit removeWorker();
}
interface FullLoadBalancer extends LoadBalancer, ResourceController {}

class RoundRobinLB(List<Worker> resources, CloudAPI cloud,
                     ResourceSpec spec, ScalingAPI scaler) implements FullLoadBalancer {

    List<Worker> available = resources;

    Unit run(){
        while (True) {
           await duration(1,1);
           scaler!workerQueueLength(length(available));
       }
    }

    Worker getWorker(){
        Worker w;
        await (available != Nil);
        w = head(available);
        available = tail(available);
        return w;
    }

    Unit releaseWorker(Worker w){
        available = appendright(available,w);
    }

    Unit addWorker(){
        // We launch a new virtual machine instance, and deploy the worker on it
        DC machine = await cloud!launchInstance(spec);
        [DC: machine] Worker w = new WorkerObject(db);
        available = appendright(available,w);
    }

    Unit removeWorker(){ // To scale down
        // Our invariant is that we have one worker per virtual machine instance
        // If the worker is available, its machine is idle and we can release the machine.
        if (available != Nil) {
            Worker w = head(available); available = tail(available);
            DC machine = await w!getDC(); cloud!releaseInstance(machine);
        }
    }

}
```

Figure 3.6: A class implementing a simple resource manager in ABS

3.4 Service-Level Agreements and Scaling

The scaling policy of the auto-scaler can be made parametric in a *service contract*. For example, it could log the ratio of deadline violations and the total number of jobs for a given custom SLA, and use the distance to the promised performance as a criterion for scaling in a similar way to the *upper* and *lower* thresholds above.

3.5 Deploying Containers on Virtual Machines

A typical problem when working with containers is the mapping between the containers and the underlying virtual machines; e.g., two containers with 2 cores each can be mapped to a virtual machines with five cores, but three containers can not be mapped to the same virtual machine. When modeling containers-based systems in ABS, it is therefore natural to use deployment components to model the containers rather than the virtual machines, and simply use a table to keep track of available resources on different virtual machines when doing container-level resource management. For example, if the workers of our running example were deployed on containers, the machines of the addWorker method of Figure 3.6 would need to find enough available resources on one of the virtual machines before the worker could be deployed on that virtual machine and made available to the application.

3.6 Example: ABS-Yarn

We have studied the modeling of dynamic resource management strategies for containers in ABS by a case study of HADOOP YARN [9], a popular MapReduce framework for big data processing with a slightly more complex resource management (see Figure 3.7). This work combines containers mapped to slave nodes with a state of the art resource management strategy. We have also used this model to study the precision of our model in terms of an empirical evaluation, comparing configurations of HADOOP YARN in ABS with benchmarks running on a cluster of 30 virtual machines. The results are reported in [14]; the paper is attached as an appendix to this deliverable.

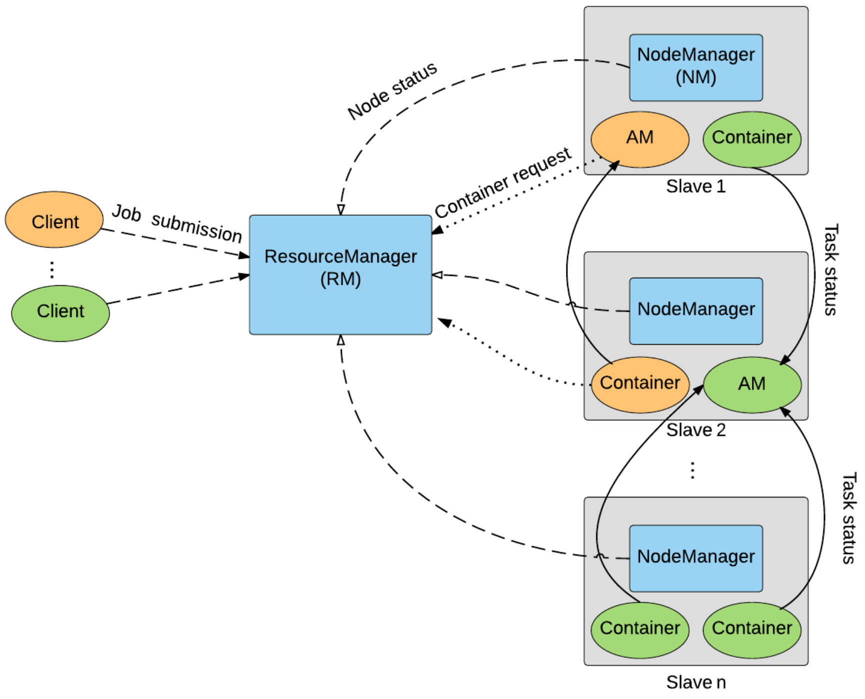


Figure 3.7: The architecture of HADOOP YARN.

Chapter 4

The ABS Cloud API

This chapter describes the facilities for modeling cloud deployment. This chapter supersedes the corresponding chapter in Deliverable D1.3.1 and presents the Cloud API as validated by the case studies.

This section builds upon Deliverable D1.2.2, which discusses resource modeling and its effects on model simulation using deployment components. This section shows support for modeling complex deployment scenarios in ABS and how this is applied in the case studies.

All ABS identifiers (classes, interfaces, functions, data types) mentioned in this chapter are either contained in the standard library or exported from the module ABS.DC if not otherwise mentioned. An import clause **import** $*$ **from** ABS.DC; in module declarations enables the code fragments to run.

4.1 Datatypes, Expressions and Resource Configurations

As mentioned in Deliverable D1.2.2, deployment components are involved in modeling resource configurations and deployment scenarios. All COGs (and their objects and processes) are deployed on some deployment component, which will restrict execution capacity according to its resource configuration. This section expands on the use of deployment components.

Finding the current deployment component. The function **thisDC**() returns a reference to the *current deployment component*, i.e., the deployment component that contains the COG on which the current process is running.

Resource Configurations. The datatype ResourceType, as described in Deliverable D1.2.2, has constructors for the resource types in use in ABS. Currently, the resource types are Speed, Cores, Bandwidth and Memory. A *resource configuration* assigns numeric values to a subset of these resource types. Resource configurations can be used to describe, create and query deployment configurations.

Additionally, the following attributes of deployment components are included in the resource configuration: Startupduration, Shutdownduration, PaymentInterval, and CostPerInterval. These attributes provide information about startup and shutdown behavior and cost accounting to CloudProvider instances.

Example:

```
def Map<Resourcetype, Rat> amazonSmallInstance() =
  map[Pair(Cores, 2), Pair(Memory, 10000),
      Pair(PaymentInterval, 5), Pair(CostPerInterval, 1)];
```

This example defines an amazonSmallInstance to be a deployment component with 2 cores and 10000 memory capacity. Note that there is no value given for bandwidth and speed; in this case, these attributes are deemed to be either infinite or not necessary for purposes of the given model. Startup and shutdown happen instantly since there is no value given for these attributes. Every 5 intervals, the cloud provider will incur 1 cost for each deployment component running with this resource configuration.

Infinite values. Cogs that are created outside any deployment component are in effect running on a deployment component with a resource configuration with infinite resources of all types. To express infinity, the module ABS.DC defines a datatype InfRat as follows:

```
data InfRat = InfRat | Fin(Rat finvalue);
```

The value of a resource can be either infinity (InfRat) or a finite value Fin(value). The concrete value can be accessed via the finvalue() function. It is an error to call finvalue on an infinite value InfRat.

4.2 Modeling Machines: the DeploymentComponent Interface

As described in Deliverable D1.2.2, COGs are deployed on deployment components (via the [DC: x] annotation to a **new** expression). A deployment component is created with a given resource configuration which influences all COGs created on that deployment component.

Example:

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

In this example, the new COG w (with an initial object of class CWorker and all objects that this object creates without annotations) will run on the deployment component dc with the resource configuration specified above.

Information about the current deployment component. The deployment component interface contains methods that give access to information about the resource configuration and current resource usage.

Example:

```
[Atomic] Rat load(Resourcetype rtype, Int periods);
[Atomic] InfRat total(Resourcetype rtype);
```

The method **load** returns a value between 0 and 100 that represents the load (consumed resources vs. available resources) for the given resource type over the last n periods. If the resource type is infinite in the resource configuration of the deployment component, the load is always 0.

The method **total** returns the total capacity of the deployment component for the given resource type. Note that the total capacity can be infinite, as in the case of an unspecified value when creating the deployment component.

Changing a resource configuration. For some simulation scenarios, it is expedient to modify the effective resource configuration. Usually these methods are called in a dedicated part of the model that implements load monitoring and resource balancing. Note that the methods in this section are sufficiently general to express a variety of theoretical and practical scenarios. For example, using Linux control groups, traffic shaping etc. it is possible to manipulate CPU, bandwidth or available memory for certain types of virtual machine or container deployments. The Cloud API supports these kinds of operations, but does not ensure that the modeled scenarios are realistic wrt. some physical deployment scenario – it is the responsibility of the modeler to ensure that models reflect the real system.

The following methods in the DeploymentComponent interface modify its resource configuration:

Example:

```
Unit incrementResources(Rat amount, Resourcetype rtype);
Unit decrementResources(Rat amount, Resourcetype rtype);
Unit transfer(DeploymentComponent target, Rat amount, Resourcetype rtype);
```

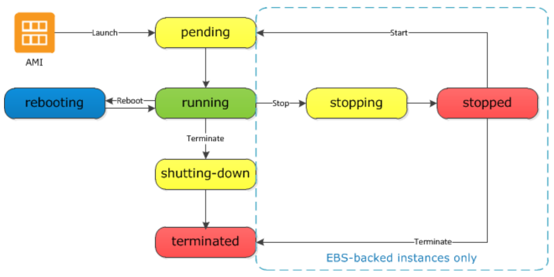


Figure 4.1: The Amazon instance lifecycle (taken from `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-lifecycle.html`)

The methods incrementResources and decrementResources increment or decrement the total available resources by the given amount. Neither have an effect when the resource type is infinite. In addition, decrementResources will not decrement below zero resources.

Incrementing and decrementing resources becomes effective in the next time slot. For example, incrementing the CPU resource type by 5 will make 5 more resources of that type available in the next and every subsequent time period.

The method **transfer** is a utility method implemented in terms of incrementing and decrementing resources. It transfers a given amount of resources to the target deployment component.

4.3 Modeling the Deployment Component Lifecycle: the CloudProvider Interface

Especially when going beyond static scenarios with a fixed number of deployment components, it is necessary to manage the creation, allocation, deallocation and destruction of deployment components. In many scenarios, the billing (cost) information is also an essential part of a model since it provides a quantitative measurement of the fitness of different deployment and balancing scenarios (see [8] for an example).

During work on the Envisage and non-Envisage case studies, it became apparent that the DeploymentComponent interface is not sufficient to properly model cloud-deployed systems. Similar solutions were re-implemented multiple times in the following areas:

- Management of a pool of deployment components;
- Management of the deployment component lifecycle;
- Modeling of accounting, tracking the cost of running machines of different sizes;
- Modeling startup and shutdown times;
- Dynamic allocation and deallocation of deployment components.

The CloudProvider interface deals with modeling the lifecycle and billing information of a number of deployment components. See Figure 4.1 for the life cycle of a deployment component that is managed by a cloud provider.

```
interface CloudProvider {
    // (pre)launchInstance might have a delay, the others are instantaneous.
    // launchInstance might hand out an already−running instance if it has
    // been released; in this case there will be no delay.
    DeploymentComponent prelaunchInstance(Map<Resourcetype, Rat> d);
    DeploymentComponent launchInstance(Map<Resourcetype, Rat> description);
    // acquireInstance, releaseInstance are called from deployment components.
    // launchInstance does the equivalent of acquireInstance.
    Bool acquireInstance(DeploymentComponent instance);
    Bool releaseInstance(DeploymentComponent instance);
    Bool shutdownInstance(DeploymentComponent instance);

    [Atomic] Rat getAccumulatedCost();
    [Atomic] Unit shutdown();

    // Instance descriptions. Call setInstanceDescriptions with a map of
    // (name −> resources) information. Then, launchInstanceNamed() returns a
    // deployment component with the specified resources, or null if the given
    // name could not be found. The resulting deployment components are then
    // handled as normal (acquire/release/kill).
    [Atomic] Unit setInstanceDescriptions(Map<String, Map<Resourcetype, Rat>> instanceDescriptions);
    [Atomic] Unit addInstanceDescription(Pair<String, Map<Resourcetype, Rat>> instanceDescription);
    [Atomic] Unit removeInstanceDescription(String instanceDescriptionName);
    [Atomic] Map<String, Map<Resourcetype, Rat>> getInstanceDescriptions();
    DeploymentComponent prelaunchInstanceNamed(String instancename);
    DeploymentComponent launchInstanceNamed(String instancename);
}
```

4.3.1 Creating and Configuring a Cloud Provider

A cloud provider instance can be obtained via a normal **new** expression. A cloud provider can be configured with a list of machine descriptions (i.e., a map of instance names to resource configurations), this enables creation of named instance types.

Example:

```
CloudProvider p = new CloudProvider("Amazon");
await p!setInstanceDescriptions(
  map[Pair("T2_MICRO", map[Pair(Memory,1), Pair(Speed,1)]),
      Pair("T2_SMALL", map[Pair(Memory,2), Pair(Speed,1)]),
      Pair("T2_MEDIUM", map[Pair(Memory,4), Pair(Speed,2)]),
      ...
      ]);
```

Backends that implement real cloud deployment of ABS code typically provide a backend- and provider-specific class that implements the CloudProvider interface and uses provider-specific API calls to instantiate physical virtual machines.

Multiple Cloud Providers

It is possible to use more than one cloud provider in a model. Each deployment component will be managed by the cloud provider that created it. The deployment component methods acquire(), release() and shutdown () will work as expected, communicating with the cloud provider. The deployment component method getProvider() will return a reference to the cloud provider that manages that deployment component.

The cloud provider of the current deployment component can be obtained via **thisDC**()!getProvider(). Note that the return value can be **null** if the current deployment component is not managed by a cloud provider.[1]

4.3.2 Launching and Acquiring Deployment Components

A deployment component is acquired from a cloud provider by calling the method launchInstanceNamed, giving the name of an instance description set via setInstanceDescriptions. It is also possible to create a deployment component by giving its resource configuration by calling the method launchInstance.

```
DeploymentComponent launchInstanceNamed(String instancename);
DeploymentComponent launchInstance(Map<Resourcetype, Rat> description);
```

[1]This is the case if the deployment component in question was created via **new**.

Upon launching a deployment component, the cloud provider starts tracking the cost (as per CostPerInterval and PaymentInterval of its resource scenario). The method will return after Startupduration, after which time the deployment component is ready to be used. In terms of lifecycle (Fig. 4.1), these methods return an instance in the state "running".

These two methods return **null** upon failure.

Pre-Launching Deployment Components

When modeling advanced dynamic deployment strategies, it is sometimes convenient to preallocate instances in preparation of higher load. The following cloud provider methods model this use case.

```
DeploymentComponent prelaunchInstanceNamed(String instancename);
DeploymentComponent prelaunchInstance(Map<Resourcetype, Rat> d);
```

The method prelaunchInstanceNamed returns a new deployment component of the given instance type. prelaunchInstance returns a deployment component matching the given resource scenario. Deployment components returned by these methods are not ready to be used. In terms of lifecycle (Fig. 4.1), these methods return an instance in the state "pending".

These instances can be used after one of the following:

- The cloud provider method acquireInstance() returns True when called with the instance as argument.
- The deployment component method acquire() returns true.
- The instance is returned by the method launchInstance() or launchInstanceNamed(). Pre-launched instances are returned by these methods if they match.

```
Bool acquireInstance(DeploymentComponent instance);
```

The acquireInstance method acquires a deployment component, i.e., after this method returns True the caller is allowed to deploys on the deployment component until the deployment component is released again. If this method returns False, the deployment component has already been acquired or is otherwise not ready to be used.

For convenience, the DeploymentComponent interface offers a convenience method Bool acquire() with the same semantics as acquireInstance. In case the deployment component is not managed by a cloud provider, this method will always return True.

4.3.3 Releasing and Shutting Down Deployment Components

A model can release a deployment component after all activities have finished.

```
Bool releaseInstance(DeploymentComponent instance);
Bool shutdownInstance(DeploymentComponent instance);
```

After releaseInstance, a subsequent call to launchInstance or launchInstanceNamed might return a reference to that same deployment component if it fits. A deployment component that has been released represents a running but idle virtual machine instance. Cost is still accrued for this instance in the cloud provider.

The DeploymentComponent interface offers a convenience method Bool release() that is equivalent to releaseInstance(). In case the deployment component is not managed by a cloud provider, release will always return True.

After calling the method shutdownInstance, no call to launchInstance will ever return a reference to that deployment component, and it will not influence the cost of running the model anymore.

The DeploymentComponent interface offers a convenience method Unit shutdown() that is equivalent to shutdownInstance().

Bibliography

[1] Pietro Abate and Schauer Johannes. Bootstrapping Software Distributions. In *CBSE'13*, 2013.

[2] Apache. Hadoop. `http://hadoop.apache.org/`.

[3] 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.

[4] Rajkumar Buyya, Chee Shin 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 Computer Systems*, 25(6):599–616, 2009.

[5] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability Modelling in the ABS Language. In *FMCO*, volume 6957 of *LNCS*, pages 204–224. Springer, 2010.

[6] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, 2014.

[7] Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. On the integration of automatic deployment into the ABS modeling language. In *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, 2015.

[8] 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.

[9] Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC'13)*, page 5. ACM, 2013.

[10] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing multi-cloud systems with CloudMF. In *NordiCloud*, volume 826, pages 38–45. ACM, 2013.

[11] Glauco Estacio Gonçalves, Patricia Takako Endo, Marcelo Anderson Santos, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *CloudCom*, 2011.

[12] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Modeling application-level management of virtualized resources in ABS. In *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*, pages 89–108, 2011.

[13] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in real-time ABS. In *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, pages 71–86. Springer, 2012.

[14] Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. ABS-YARN: A formal framework for modeling Hadoop YARN clusters. In Perdita Stevens and Andrzej Wasowski, editors, *19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*, volume 9633 of *Lecture Notes in Computer Science*. Springer-Verlag, 2016.

[15] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html`.

[16] Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. Minisearch: A solver-independent meta-search language for minizinc. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2015.

Glossary

ABS Abstract Behavioural Specification language. An executable class-based, concurrent, object-oriented modelling language based on Creol, created for the HATS project. In Envisage this language has been extended with the notion of deployment component, which is a container providing running objects with the needed resources.

ABS Cloud API An interface included in the ABS Standard Library for the modeling of typical remote calls to a cloud infrastructure to acquire, release, monitor and manage virtual computing resources.

ABS Standard Library It includes ABS class and interface declarations that are typically included into ABS programs.

API Application Programming Interface. It usually identifies the set of external remote calls a program or a service exposes to its clients.

Cloud Computing metaphor identifying utilities for acquisition and consumption of virtual computing resources on-demand.

Cloud Application Deployment The deployment of an application on a cloud infrastructure corresponds to the definition of the software architecture of the application (software artifacts and their relationship) and the description of the distribution of the software components on top of the computing resources offered by the clod infrastructure.

Deployment component ABS abstraction encoding virtual machine identities: objects executing inside a specific deployment component corresponds to processes running in a specific virtual machine and using its resources and computation power.

Dynamic deployment Acquisition or release of new resources like computing power, memory, etc. during a computing system lifetime, including the allocation of the new acquired resources to corresponding computing components.

Static deployment Initial configuration of a component-based computing system obtained by means of the proper distribution and interconnection of the components over the available computing resources.

Appendix A

Declarative Elasticity in ABS

Declarative Elasticity in ABS[⋆]

Stijn de Gouw[1], Jacopo Mauro[3], Behrooz Nobakht[2], and Gianluigi Zavattaro[4]

[1] Fredhopper, Netherlands
[2] Leiden University, Netherlands
[3] University of Oslo, Norway
[4] University of Bologna/INRIA, Italy

Abstract. Traditional development methodologies that separate software design from application deployment have been replaced by approaches such as continuous delivery or DevOps according to which deployment issues should be taken into account already at the early stages of development. This calls for the definition of new modeling and specification languages. In this paper we show how deployment can be added as a first-class citizen in the object-oriented specification language ABS. We follow a declarative approach: programmers specify deployment constraints and a solver synthesizes ABS classes exposing methods like `deploy` (resp. `undeploy`) that executes (resp. cancels) configuration actions changing the current deployment towards a new one satisfying the programmer's desiderata. Differently from previous works, this novel approach allows for the specification of dynamic modifications thus supporting the declarative modeling of elastic applications.

1 Introduction

Software applications deployed and executed on cloud computing infrastructures should flexibly adapt by dynamically acquiring or releasing computing resources. This is necessary to properly deliver to the final users the expected services at the expected level of quality, maintaining an optimized usage of the computing resources. For this reason, modern software systems call for novel engineering approaches that anticipate the possibility to reason about deployment already at the early stages of development.

Modeling languages like TOSCA [21], CloudML [16], and CloudMF [13] have been proposed to specify the deployment of software artifacts, but they are mainly intended to express deployment of already developed software. An actual integration of deployment in software development is still far from being obtained in the current practices. To cover this gap, in this paper we address the problem of extending the ABS (Abstract Behavioural Specification) language [2]

[⋆] Supported by the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`) and H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems* (`http://www.hyvar-project.eu`).

with linguistic constructs and mechanisms to properly specify deployment. Following [9] our approach is declarative: the programmer specifies deployment constraints and a solver computes actual deployments satisfying such constraints. In previous work [10] we presented an external engine able to synthesize ABS code specifying the initial static deployment; in this paper we fully integrate this approach in the ABS language allowing for the declarative specification of the dynamic upscale/downscale of the modeled application depending, e.g., on the monitored workload or the current level of resource usage.

ABS is an object-oriented modeling language with a formally defined and executable semantics. It includes a rich tool-chain supporting different kinds of analysis (like, e.g., logic-based modular verification [11], deadlock detection [15], and cost analysis [3]). Production code can be automatically obtained from ABS specifications by means of code generation. ABS has been mainly used to model systems based on asynchronously communicating concurrent objects, distributed over Deployment Components corresponding to containers offering to objects the resources they need to properly run. For our purposes, we adopted ABS because it allows the modeling of computing resources and it has a real-time semantics reflecting the way in which objects consume resources. This makes ABS particularly suited for modeling and reasoning about deployment.

Our initial proposal for the declarative modeling of deployment into ABS [10] was based on three main pillars: (i) classes are enriched with annotations that indicate functional dependencies of objects of those classes as well as the resources they require, (ii) a separate high-level language for the declarative specification of the deployment, (iii) an engine that, based on the annotations and the programmer's requirements, computes a fully specified deployment that minimizes the total cost of the system. The computed deployment is expressed in ABS and can be manually included in a main block.

The work in [10] had two main limitations: (i) there was no way to express dynamic deployment decisions like, e.g., the need to upscale or downscale the modeled system and (ii) there was no real integration of the code synthesized by the engine in the corresponding ABS specification. In this paper we address these limitations by promoting the notion of deployment as a first-class citizen of the language. During a pre-processing phase, the new tool SmartDepl generates ad-hoc classes exposing the methods `deploy` and `undeploy` to dynamically upscale and downscale the system. The deployment requirements can now also reuse already deployed objects just specifying which existing objects could be used, and how they should be connected with new objects to be freshly deployed. This has been the fundamental step forward that allowed us to support dynamic modification of the current deployment. Moreover, other relevant contributions of this paper are (i) a more natural high-level language for the specification of requirements that now supports universal and existential quantifiers, and (ii) the usage of the delta modules and the variability modeling features of the ABS framework [7] to automatically and safely inject the deployment instructions into the existing ABS code.

Our ABS extension and the realization of the corresponding SmartDepl tool have been driven and validated against the Fredhopper Cloud Services, an industrial case-study of the European FP7 Envisage project. The Fredhopper Cloud Services offer search and targeting facilities on a large product database to e-Commerce companies. Depending on the specific profile of an e-Commerce company Fredhopper has to decide the most appropriate customized deployment of the service. Currently, such decisions are taken manually by an operation team which decides customized, hopefully optimal, service configurations taking into account the tension among several aspects like the level of replications of critical parts of the service to ensure high availability. The operators manually perform the operations to scale up or down the system and this usually causes the over-provision of resources for guaranteeing the proper management of requests during a usage peak. With our extension of ABS, we have been able to realize a new modeling of the Fredhopper Cloud Services in which both the initial deployment and the subsequent up- and down-scale is expected to be executed automatically. This new model is a first fundamental step towards a new more efficient and elastic deployment management of the Fredhopper Cloud Services.

Structure of the paper Section 2 describes the Fredhopper Cloud Services case-study. Section 3 reports the ABS deployment annotations that we already defined in [10]. Section 4 presents the new high-level language for the specification of deployment requirements while Section 5 discusses the corresponding solver. Finally, the application of our technique to the Fredhopper Cloud Services use-case is reported in Section 6. Concluding remarks are in Section 7.

2 The Fredhopper Cloud Services

Fredhopper uses the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). The Fredhopper Cloud Services drives over 350 global retailers with more than 16 billion in online sales every year. A customer (service consumer) of Fredhopper is a web shop, and an end-user is a visitor of the web shop.

The services offered by Fredhopper are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Typically, software services are deployed as *service instances*. Each instance offers the same service and is exposed via the Load Balancing Service, which in turn offers a service endpoint. Load Balancers serve as endpoints and distribute requests over the service instances. Figure 1 shows a block diagram of the Fredhopper Cloud Services.

The number of requests can vary greatly over time, and typically depends on several factors. For instance, the time of the day in the time zone where most of the end-user are plays an important role (typical lows in demand are observed between 2 am and 5 am). Figure 2 is a typical real-world graph for a single day (with data up to 18:00) showing the number of queries per second (y-axis,

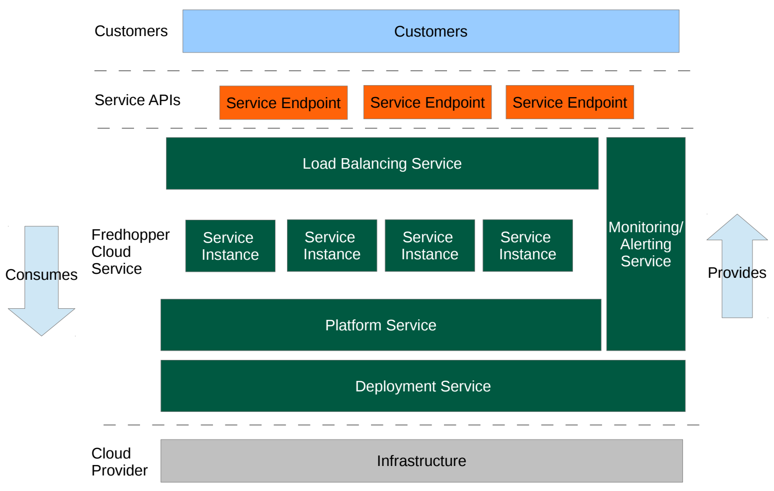


Fig. 1. The architecture of the Fredhopper Cloud Services

ranging from 0-25 qps, the horizontal dotted lines are drawn at 5,10,15 and 20 qps) over the time of the day (x-axis, starting at midnight, the vertical dotted lines indicate multiples of 2 hours). The 2a - 5am low is clearly visible.

Peaks can occur during promotions of the shop or around Christmas. To ensure a high quality of service, web shops negotiate an aggressive Service Level Agreement (SLA) with Fredhopper. QoS attributes of interest include query latency (response time) and throughput (queries per second). For example, based on the negotiated SLA with a customer, services must maintain 100 queries per seconds with less than 200 milliseconds of response time over 99.5% of the service uptime, and 99.9% with less than 500 milliseconds.

Previous work reported in [10] aimed to compute an optimal initial deployment configuration (based on the size of the product catalogue, number of expected visitors and cost of the required virtual machines). The computation was based on an already available model of the Fredhopper Cloud Services written in the ABS language. In this paper we address the problem of maintaining a high quality of service after this initial set-up by taking dynamic factors into account, such as fluctuating user-demand and unexpectedly failing virtual machines.

The solution that we propose is based on a tool named SmartDepl that, when integrated in the ABS specification of the Fredhopper Cloud Services, enables the modeling of the automatic dynamic upscaling or downscaling. When the decision to scale up or down is made, SmartDepl indicates how to automatically evolve the deployment configuration. This is not a trivial task: the desired deployment configuration should satisfy various requirements, and those can trigger the need to instantiate multiple service instances that furthermore require proper configuring to ensure they function correctly.

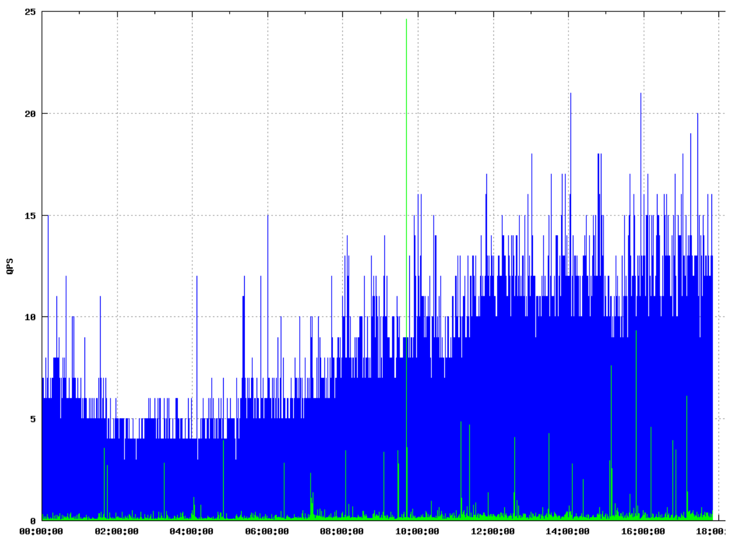


Fig. 2. Number of queries per second (in green the query processing time).

The requirements can originate from both business decisions or technical reasons. For instance, for security reasons, services that operate on sensitive customer data should not be deployed on machines shared by multiple customers. Below we list some of these requirements.

- To increase fault-tolerance, we aim to spread virtual machines across geographical locations. Amazon allows specifying the desired region (a geographical area) and availability zone (a geographical location in a region) for a virtual machine. Fault tolerance is then increased by balancing the number of machines between different availability zones. Thus, when scaling, the number of machines should be adjusted in all zones simultaneously. Effectively this means that with two zones, we scale up or down with an even number of machines.
- Each instance of a Query service is in one of two modes: 'live' mode to serve queries, or 'staging' mode to serve as an indexer (i.e., to publish updates to the product catalogue). There always should be at least one instance of Query service in staging mode.
- The network throughput and latency between the PlatformService and indexer is important. Since the infrastructure provider gives better performance for traffic between instances in the same zone, we require the indexer and PlatformService to be in the same zone.
- Installing an instance of the QueryService requires the presence of an instance of the DeploymentService on the same virtual machine.
- For performance reasons and fault tolerance, load balancers require a dedicated machine without other services co-located on the same virtual machine.

3 Annotated ABS

The ABS language is designed to develop executable models. It targets distributed and concurrent systems by means of concurrent object groups and asynchronous method calls. Here, we will recap just the specific linguistic features of ABS to support the modeling of the deployment; for more details we refer the interested reader to the ABS project website [2] and [10] for the cost annotations.

The basic element to capture the deployment in ABS is the *Deployment Component* (DC), which is a container for objects/services that, intuitively, may model a virtual machine running those objects/services. ABS comes with a rich API that allows the user to model a cloud provider of deployment components.

```
1 CloudProvider cProv = new CloudProvider("Amazon");
2 cProv.addInstanceDescription(Pair("c3",
3   InsertAssoc(Pair(CostPerInterval,210),
4     InsertAssoc(Pair(Memory,7500),
5       InsertAssoc(Pair(Cores,4), EmptyMap)))));
6 DeploymentComponent dc = cProv.prelaunchInstanceNamed("c3");
7 [DC: dc] Service s = new QueryServiceImpl();
```

In the ABS code above, the cloud provide “Amazon” is modeled as the object `cProv` of type `CloudProvider`. The fact that “Amazon” can provide a virtual machine of type “c3” is captured by calling the `addInstanceDescription` in Line 2. With this instruction we also specify that c3 virtual machines cost 210 cents an hour and provide 7.5 GB of RAM and 4 cores. In Line 5 an instance of “c3” is launched and the corresponding deployment component is saved in the variable `dc`. Finally, in Line 6, a new object of type `QueryServiceImpl` (implementing interface `Service`) is created and deployed in the deployment component `dc`.

In ABS it is possible to declare interface hierarchies and define classes implementing them.

```
interface Service { ... }
interface IQueryService extends Service { ... }
class QueryServiceImpl(DeploymentService ds, Bool staging)
  implements IQueryService { ... }
```

In the excerpt of ABS above, the `IQueryService` service is declared as an interface that extends `Service`, and the class `QueryServiceImpl` is defined as an implementation of this interface. Notice that the initialization parameters required at object instantiation are indicated as parameters in the corresponding class definition.

Classes can be annotated with annotations that denote the cost and the requirements of an object of that class.

```
[Deploy: scenario[Name("staging"), Cost("Cores", 2),
 Cost("Memory",7000), Param("staging", Default("True")),
 Param("ds", Req)] ]
[Deploy: scenario[Name("live"), Cost("Cores", 1),
 Cost("Memory",3000), Param("staging", Default("False")),
 Param("ds", Req)] ]
```

```
b_expr : b_term (bool_binary_op b_term )* ;
b_term : ('not')? b_factor ;
b_factor : 'true' | 'false' | relation ;
relation : expr (comparison_op expr)? ;
expr : term (arith_binary_op term)* ;
term : INT                                          |
  ('exists' | 'forall') VARIABLE 'in' type ':' b_expr |
  'sum' VARIABLE 'in' type ':' expr                  |
  (( ID | VARIABLE | ID '[' INT ']' ) '.')? objId      |
  arith_unary_op expr                                |
  '(' b_expr ')'                                     ;
objId :  ID | VARIABLE | ID '[' ID ']' | ID '[' RE ']';
type : 'obj' | 'DC' | RE ;
bool_binary_op : 'and' | 'or' | 'impl' | 'iff' ;
arith_binary_op : '+' | '-' | '*' ;
arith_unary_op : 'abs' ; // absolute value
comparison_op : '<=' | '=' | '>=' | '<' | '>' | '!=' ;
```

Table 1. DRL grammar.

The previous two annotations describe two possible deployment scenarios for an object of the class `QueryServiceImpl`. The first annotation captures the deployment of a Query Service in staging modality while the second captures the deployment of the Query Service in live modality. A Query Service in staging modality requires 2 cores and 7GB of RAM while in live mode it only requires 1 core and 3GB of RAM. The creation of a Query Service Object requires an object of type `DeploymentService` that is associated with the parameter `dc` while the parameter `staging` is set to true or false according to the desired deployment scenario.

4 The Declarative Requirement Language DRL

When a system deployment is automatically computed, a user expects to reach specific goals and could have some desiderata. For instance, in the considered Fredhopper Cloud Services use case, the initial goal is to deploy a given number of Query Services and a Platform Service, possibly located on different machines (e.g., to improve fault tolerance) and later on to upscale or downscale the system according to the monitored traffic.

All these goals and desiderata can be expressed in the *Declarative Requirement Language* (DRL): a new language for stating the constraints that the final configuration should satisfy.

As shown in Table 1 that reports the DRL grammar defined using the ANTLR tool,[5] a desiderata is a boolean formula `b_expr` obtained using the usual logical

[5] ANTLR (ANother Tool for Language Recognition) - http://www.antlr.org/

connectives over comparison of arithmetic expression. The atom of this arithmetic expression may be integers (Line 6), a quantifier statement (Line 7), a sum statement (Line 8) and an identifier for the number of deployed objects (Line 9). The number of object deploy using a given scenario is defined by its class identifier and the scenario name enclosed in square brackets (Line 14). As an example the following formula requires the presence of at least an object of class `QueryServiceImpl` deployed in staging mode.

```
QueryServiceImpl[staging] > 0
```

The square brackets may be omitted (Line 12 - first option) for objects that have only one default deployment scenario. Regular expression (`RE` in Line 12) can be used to match objects deployed using different deployment scenarios. The number of deployed objects can also be prefixed by a deployment component identifier to denote just the number of objects defined within that specific deployment component. As an example, the deployment of only one object of class `DeploymentServiceImpl` on the first and second instance of a "c3" virtual machine can be enforced as follows.

```
c3[0].DeploymentServiceImpl = 1 and
  c3[1].DeploymentServiceImpl = 1
```

Here the 0 and 1 numbers between the square brackets represent respectively the first and second virtual machine of type "c3" . To shorten the notation, the `[0]` can be omitted (Line 9).[6]

It is possible to use also quantifiers and sum expressions to capture more concisely some of the desired properties. Variables are identifiers prefixed with a question mark. As specified in Line 15, quantifiers and sum term variables can range on all the objects (`'obj'`), all the deployment components (`'DC'`), or just on all the virtual machines matching a given regular expression (`RE`). In this way it is possible to express more elaborate constraints such as the co-location or distribution of objects, or limit the amount of objects deployed on a given DC.[7] As an example, to enforce the constraint that every Query Service requires a Deployment Service installed on its virtual machine we can require the following.

```
forall ?x in DC: (
  ?x.QueryServiceImpl['.*'] > 0  impl
  ?x.DeploymentServiceImpl > 0)
```

Here we use the regular expression `'.*'` to be able to match with only one repetition the Query Services deployed in `staging` and `live` mode.

Finally, requiring for instance the load balancer to be installed alone in a virtual machine can be done as follows.

---

[6] We assume that the user could launch only a bounded number of deployment components. In particular, for every cloud deployment type SmartDepl allows to specify the maximal number of deployment components that can be created.

[7] DRL improves on the specification language presented in [10] because the addition of the quantifiers and the sum terms allows the user to write her desiderata in a more concise and natural way.

```
 1 { "id": "AddQueryDeployer",
 2   "specification": "QueryServiceImpl[live] = 1",
 3   "obj": [ { "name": "platformObj",
 4              "provides": [ {
 5                "ports": [ "MonitorPlatformService",
 6                           "PlatformService" ],
 7                "num": -1 } ],
 8              "interface": "PlatformService" },
 9            { "name": "loadBalancerObj",
10              "provides": [ {
11                "ports": [ "LoadBalancerService" ],
12                "num": -1 } ],
13              "interface": "LoadBalancerService" },
14            { "name": "serviceProviderObj",
15              "provides": [ {
16                "ports": [ "ServiceProvider" ],
17                "num": -1 } ],
18              "interface": "ServiceProvider" } ],
19   "DC": [] }
```

Table 2. JSON annotation example.

```
forall ?x in DC: (
  ?x.LoadBalancerServiceImpl > 0 impl
  (sum ?y in obj: ?x.?y) = ?x.LoadBalancerServiceImpl)
```

5 Deployment Engine

SmartDepl is the tool that we have implemented to support the user with the automation of deployment choices. The key idea of SmartDepl is to allow the user to specify declaratively what she wants to deploy and write the program abstracting from the concrete deployment decision. More concretely, we require the user to specify all its deployment needs as program annotations. SmartDepl processes them and it generates for every deployment need a new class that specifies the deployment steps to reach the desired target. This class can be used by the user to trigger the execution of the deployment but also to undo it in case the system needs to downscale.

As an example, imagine that an initial deployment of the Fredhopper Cloud Services has been already obtained and that, based on a monitor decision, the user wants to add 1 Query Services in live mode. The annotation required by SmartDepl for capturing this need is the JSON object defined in Table 2.

In Line 1, by using the keyword `"id"` the user specifies that the name of the class containing the deployment code is `AddQueryDeployer`. As we will see later, the user can exploit this name to upscale and downscale the system assuming the class existence. The second line contains the desired configuration in DRL.

By using the keyword `"obj"`, Lines 3-18 define objects that are assumed to be already available, hence are not needed to be re-deployed. Assuming that the user has already a working Fredhopper Cloud Services, she knows indeed that there is already a Platform Service, a Load Balancer and a Service Provider deployed. Every available object is defined by assigning to it a unique name (the keyword `"name"` in Lines 3,9,14), the interfaces it provide (the keyword `"port"` in Lines 5-6,11,16) with the amount of services that can use them (keyword `"num"` in Lines 7,12,17 — in this case a -1 value means that the object can be used by an unbounded number of other objects), and the object interface (keyword `"interface"` in Lines 8,13,18). Finally, by using the keyword `"DC"` the user can also specify if there are some existing deployment components with free resources that can be used to deploy new objects inside them. In this case, for fault tolerance reasons the user wants to deploy the Query Service in new machines and therefore the `"DC"` is left empty (Line 19).

For the time being this annotation is given in a textual form but in order to better support the user we are considering its generation via a graphical notation. For the interested reader, the formal specification of the JSON annotation is defined in https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/spec/smart_deploy_annotation_schema.json.

Once the annotation is given the user may freely use this class. For instance, the ABS snipped code to upscale and downscale the system based on a monitor decision follows.

```
1  while ( ... ) {
2   if ( monitor.scaleUp() ) {
3      SmartDeployInterface depObj = new AddQueryDeployer(
4        cProv, platformService, loadBalancerService, serviceProvider);
5      depObj.deploy();
6      depObjList = Cons(depObj,depObjList);
7    } else if ( (monitor.scaleDown()) && (depObjList != Nil) ) {
8       SmartDeployInterface depObj = head(depObjList);
9       depObjList = tail(depObjList);
10      depObj.undeploy(); } }
```

The idea is to store the references to deployment decisions in a list called depObjList. When the monitor decides to upscale by adding new Query Services (Line 2) a new deployment decision object is created (Line 3). In this case AddQueryDeployer is the name associated with the annotation previously discussed. Its first parameter is the cloud provider, as defined for instance in Section 3. The following parameters are the objects already available for the deployment that do not need to be re-deployed from scratch. These are given according to the order they are defined in the annotation in Table 2. The interface of this class is SmartDeployInterface which is initially an empty interface that SmartDepl populated with: i) a deploy method to realise the deployment of the desired configuration, ii) an undeploy method to undo the deployment gracefully by removing the virtual machine created with the application of the deploy method, iii) getter methods to retrieve the new list of objects and deployment components created by running the deploy method (e.g., to retrieve the list of all the Query Service created by depObj.deploy() it is possible to call the operation depObj.getIQueryService()). The real addition of the Query Service is performed in Line 5 with the call of the deploy method. If instead the monitor decides to downscale (Line 7), the last deployment solution is retrieved (Line 8) and then the action performed by the deployment are undone by calling the undeploy method.[8]

Technically, SmartDepl is written in python (∼1k lines of code) and relies on Zephyrus2, a configuration optimizer that given the user desiderata and a universe of components is able to compute the optimal configuration satisfying the user needs.[9] SmartDepl uses the cost annotations as defined in Section 3 to compute a configuration that satisfies the user requirements minimizing the cost of the deployment components that need to be created and, in case of ties, minimizing also the number of created objects. Once a configuration is obtained, SmartDepl uses a topological sort to take into account all the object dependencies and establishes the correct sequence of deployment instructions to realise the computed configuration. SmartDepl generates the code of the classes and the methods to inject to the interface exploiting Delta Model techniques [7]. SmartDepl notifies the user in case no configuration can satisfy the desiderata, e.g., when the specification is too restrictive. Moreover, SmartDepl also notifies the user when it is unable to generate a sequence of deployment actions due to mutual dependencies between the objects.[10]

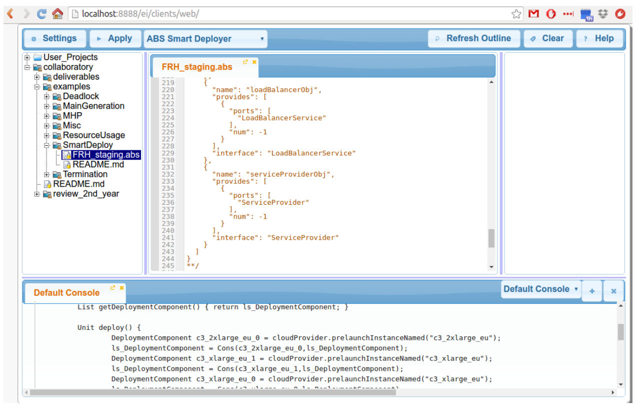
As an example the deploy code generated by SmartDepl for the annotation defined in Table 2 is the following.

```
Unit deploy() {
  DeploymentComponent c3_0 = cloudProvider.prelaunchInstanceNamed("c3");
  ls_DeploymentComponent = Cons(c3_0,ls_DeploymentComponent);
  [DC: c3_0] DeploymentService oDef___DeploymentServiceImpl_0_c3_0 =
    new DeploymentServiceImpl(platformObj);
  ls_DeploymentService = Cons(oDef___DeploymentServiceImpl_0_c3_0,
    ls_DeploymentService);
  [DC: c3_0] IQueryService olive___QueryServiceImpl_0_c3_0 = new
    QueryServiceImpl(oDef___DeploymentServiceImpl_0_c3_0, False);
  ls_IQueryService = Cons(olive___QueryServiceImpl_0_c3_0, ls_IQueryService);
  ls_Service = Cons(olive___QueryServiceImpl_0_c3_0, ls_Service);
  ls_EndPoint = Cons(olive___QueryServiceImpl_0_c3_0, ls_EndPoint);
}
```

[8] Since ABS does not have an explicit operation to force the removal of objects the undeploy procedure just removes the references to these objects leaving the garbage collector to actually remove them. The deployment components created by the deploy methods are removed instead using an explicit kill primitive provided by ABS.

[9] SmartDepl uses Zephyrus2 (freely available at https://jacopomauro@bitbucket.org/jacopomauro/zephyrus2.git) since it allows the use of a new expressive language and because it relies on MiniSearch [24], a new efficient and flexible framework for planning the search strategies. Zephyrus2 is a completely new re-engineering of the previous Zephyrus solver [8,9].

[10] This occurs when the creation of an object requires the execution of a complex protocol like for instance what happens for the boostrapping of Linux distribution [1].



```
    {
      "name": "loadBalancerObj",
      "provides": [
        {
          "ports": [
            "LoadBalancerService"
          ],
          "num": -1
        }
      ],
      "interface": "LoadBalancerService"
    },
    {
      "name": "serviceProviderObj",
      "provides": [
        {
          "ports": [
            "ServiceProvider"
          ],
          "num": -1
        }
      ],
      "interface": "ServiceProvider"
    }
  ]
}
**/
```

```
    List getDeploymentComponent() { return ls_DeploymentComponent; }

    Unit deploy() {
        DeploymentComponent c3_2xlarge_eu_0 = cloudProvider.prelaunchInstanceNamed("c3_2xlarge_eu");
        ls_DeploymentComponent = Cons(c3_2xlarge_eu_0,ls_DeploymentComponent);
        DeploymentComponent c3_xlarge_eu_1 = cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
        ls_DeploymentComponent = Cons(c3_xlarge_eu_1,ls_DeploymentComponent);
        DeploymentComponent c3_xlarge_eu_0 = cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
```

Fig. 3. SmartDepl execution within the ABS toolchain IDE.

In the previous code, at Line 3, a new deployment component c3_0 is created. At Lines 4-5 an object of class DeploymentService is created. This is due to the fact that every Query Service requires its Deployment Service (i.e., it is one of the required parameters, cfr. Section 3) and therefore this object needs to be created and deployed before the Query Service. In Lines 8-9 the desired object of class IQueryService is finally created. Both the objects are deployed on c3_0.

Even though for the presentation sake this is just a simple example, it is immediately possible to notice that by using SmartDepl the user is alleviated from the burden of the deployment decisions. Indeed, she can specify the desired configuration without taking care of the dependencies of the various objects and their distributed placement for obtaining the cheapest possible solution.

SmartDepl is open source and freely available from https://github.com/jacopoMauro/abs_deployer/tree/smart_deployer. To increase its portability it can be installed also by using the Docker container technology [12]. As shown in Figure 3, SmartDepl has also been integrated into the ABS toolchain,[11] i.e., an IDE and a collection of tools for writing, inspecting, checking, and analyzing ABS programs developed withing the Envisage European project.

6 Application to the Fredhopper use case

In this section we report about the modeling with SmartDepl of the concrete deployment requirements of the Fredhopper Cloud Services, previously introduced in Section 2. We decided to apply our techniques to the Fredhopper Cloud Services use case because it was already modeled in ABS and, thanks to a profiling activity of the real system, the cost of the services are known.

SmartDepl was used twice: to synthesize the initial static deployment of the entire framework and for dynamically adding (and then removing) single in-

[11] http://abs-models.org/installation/

stances of Query Services in case the system needs to scale (up or down). Since Fredhopper Cloud Services is using Amazon EC2 Instance Types we used two types of deployment components corresponding to the “xlarge” and “2xlarge” instances of the Compute Optimized instances (version 3)[12] of Amazon. Moreover, for fault tolerance and stability, Fredhopper Cloud Services uses instances in multiple regions in Amazon (regions are geographically separate, so even if there is a force majeure in one region, other regions are not necessarily affected). We model the instance types in different regions as follows: “c3_xlarge_eu”, “c3_xlarge_us”, “c3_2xlarge_eu”, “c3_2xlarge_us” (“eu” refers to a European region, “us” refers to an American region).

For the static deployment of the system Fredhopper Cloud Services requires the deployment of a Load Balancer, a Platform Service, a Service Provider, 2 Query Services among whom at least one in staging mode. This can be easily expressed as follows.

```
LoadBalancerServiceImpl = 1 and PlatformServiceImpl = 1 and
ServiceProviderImpl = 1 and QueryServiceImpl[staging] > 0 and
QueryServiceImpl[staging] + QueryServiceImpl[live] = 2
```

For the correct functioning of the system a Query Service requires a Deployment Service installed on the same machine. This constraint can be expressed as shown in Section 4. The requirement that a ServiceProvider is present on every machine containing a Platform Service can be expressed as follows.

```
forall ?x in DC: (?x.PlatformServiceImpl > 0 impl ?x.ServiceProviderImpl > 0)
```

Not all services can be freely installed on an arbitrary virtual machine. To increase fault tolerance Fredhopper Cloud Services requires that the Load Balancer, the Query/Deployment Services, and the Platform Service/Service Provider are never co-located on the same virtual machine. This can be easily expressed as shown at the end of Section 4.

To cope with catastrophic failures, as discussed in Section **??**, Fredhopper Cloud Services distribute the Query Services among the available regions. This can be enforced by constraining the number of the Query Services in the different data centers to be equal. In DRL this can be expressed using regular expressions as follows.

```
(sum ?x in '.*_eu': ?x.QueryServiceImpl['.*']) =
(sum ?x in '.*_us': ?x.QueryServiceImpl['.*'])
```

As described in Section 4, for performance reasons, the Query Service in Staging mode should be located in the zone of the Platform Service, since Amazon connects instances in the same region with low-latency links. For the European data-center this can be expressed by:

```
(sum ?x in '.*_eu': ?x.QueryServiceImpl[staging]) > 0) impl
(sum ?x in '.*_eu': ?x.PlatformServiceImpl ) > 0)
```

[12] https://aws.amazon.com/ec2/instance-types/

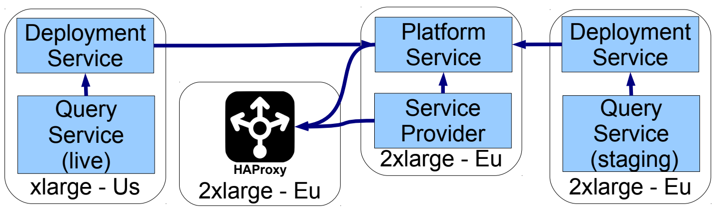


Fig. 4. Example of automatic objects allocation to deployment components.

With this specification, SmartDepl is able to compute that the initial configuration that minimizes the total costs per interval is the one depicted in Figure 4 that uses three "2xlarge" instances in Europe for deploying the Load Balancer, the Platform Service, one the staging Query Service, and one "xlarge" instance in US to deploy the Query Service in live mode.

After this initial deployment, the Cloud engineers of Fredhopper Cloud Services rely on feedback provided by monitors to decide if more Query Services in live mode are needed. Figure 5 and 6 show some of the main metrics for a single customer used to determine the scaling. The timescale in the figures is 1 day, but this can be adjusted to see trends over a longer period, or zoom in on a shorter period. The figures show that the number of queries served per second (qps, first graph of Figure 5) is relatively high and the requests (second graph of Figure 5) are relatively low, so requests are not queuing. Furthermore the CPU usage (third graph of Figure 5) and memory consumption with small swap space used (second and third graphs of Figure 6) look healthy. Hence, no scaling is needed.

If we would have needed to scale up, two Query Service instances are added (one in an EU region, and one in an US region for balancing across regions). On the other hand, if there is unnecessary overcapacity, the most recent ones can be shut down. However, since the decision to scale is a manual process by the Cloud operations team, and Fredhopper has very aggressive SLAs, the operations team is typically conservative with downscaling, leading to potential over-spending. The ability of SmartDepl to deploy in the programming language (ABS) itself allows to put auto-scaling on a rigorous basis by its tight integration into the ABS monitoring framework.

Furthermore, while the operations team currently use ad-hoc scripts to configure newly added or removed service instances, and these scripts are specific to the infrastructure provider, SmartDepl automatically generates code that accomplishes this (for example, see Table 2). SmartDepl is flexible in the sense that it is infrastructure independent, allowing to seamlessly switch between different infrastructure providers: virtual machines are launched and terminated through a generic API offered by ABS for managing virtual resources, and this API is implemented for different infrastructure providers (Amazon, Docker, OpenStack).

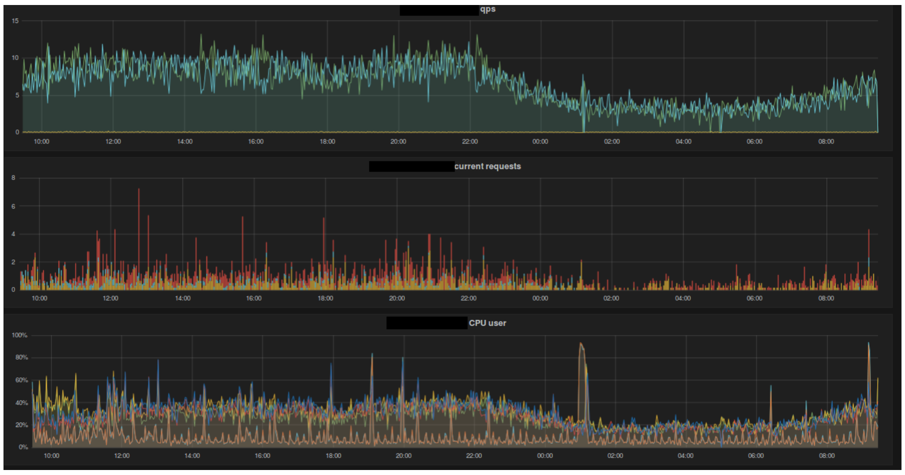


Fig. 5. Metrics graphed over a single day for a customer (a).

To automatically generate the desired deployment configuration, SmartDepl uses as specification all the previous constraints except that now instead of requiring a Platform Service and a Load Balancer we simply require two Query services in live mode. In this case, as expected after the deployment of the initial framework, the best solution is to deploy one Query Service in Europe and one in US using "xlarge" instances.

SmartDepl is able to compute the optimal deployment configurations and generate the code in less than 5 seconds. The ABS model used with all the annotations and specifications is available at https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/test/FRH_staging.abs

7 Related Work and Conclusions

We have presented an extension of the ABS specification language supporting the modeling of deployment following a declarative approach: the programmer specifies deployment constraints, and a solver synthesizes ABS classes including methods that executes deployment actions able to reach an optimal application configuration that satisfies the given constraints. Our approach, that takes inspiration from [9] and significantly improves our initial work [10], could be easily applied to any other object-oriented language offering primitives for the acquisition and release of computing resource.

Many management tools for the bottom-up deployment such as CFEngine [6], Puppet [19], MCollective [23], and Chef [22] exists. Such tools allow for the declaration of components, by indicating how they should be installed on a given machine, together with their configuration files, but they are not able to automatically decide where components should be deployed and how to interconnect

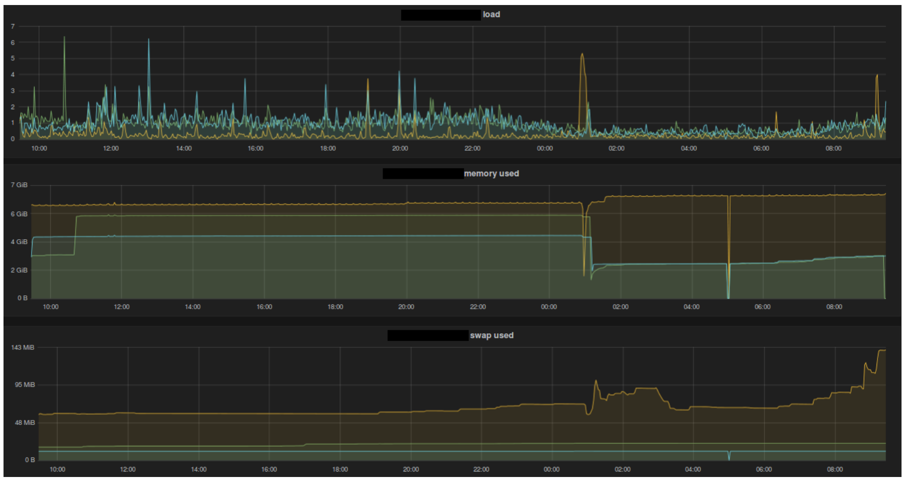


Fig. 6. Metrics graphed over a single day for a customer (b).

them for an optimal resource allocation. The alternative holistic approach allows for the modeling of the entire application and the deployment plan is then derived in a top-down manner. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [21]. Following a similar philosophy, we can mention Terraform [17], Apache Brooklyn [4], and other tools supporting the Cloud Application Management for Platforms protocol [20]. In [5] a first attempt of combination of the holistic and the bottom-up approaches is reported: a global deployment plan expressed in TOSCA is checked for correctness against local specifications of the deployment lifecycle of the single components.

Similarly to our approach, ConfSolve [18] and Engage [14] use a solver to plan deployment starting from the local requirements of components, but these approaches were not incorporated in fully-fledged specification languages (including also behavioral descriptions as in our case with ABS).

As a future work we plan to investigate the possibility to invoke at run time the external deployment engine. In this way, it could be possible to dynamic re-define the deployment constraints by means of a dynamic tuning of the engine. Nevertheless, dynamically computing the deployment steps may require additional elements such as the support of new reflection primitives to get a snapshot of the running application, and possibly the use of sub-optimal solutions when computing the optimal configuration takes too much time.

References

1. P. Abate and S. Johannes. Bootstrapping Software Distributions. In *CBSE'13*, pages 131–142. ACM, 2013.

2. Abstract behavioral specification language. http://www.abs-models.com/.
3. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: static analyzer for concurrent objects. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.
4. Apache Software Foundation. Apache Brooklyn. https://brooklyn.incubator.apache.org/.
5. A. Brogi, A. Canciani, and J. Soldani. Modelling and analysing cloud application management. In *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 19–33. Springer, 2015.
6. M. Burgess. A Site Configuration Engine. *Computing Systems*, 8(2), 1995.
7. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability Modelling in the ABS Language. In *FMCO*, volume 6957 of *LNCS*, pages 204–224. Springer, 2010.
8. R. D. Cosmo, M. Lienhardt, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Automatic Application Deployment in the Cloud: from Practice to Theory and Back. In *CONCUR*, volume 42 of *LIPIcs*, pages 1–16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
9. R. D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, 2014.
10. S. de Gouw, M. Lienhardt, J. Mauro, B. Nobakht, and G. Zavattaro. On the integration of automatic deployment into the ABS modeling language. In *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, 2015.
11. C. C. Din, R. Bubel, and R. Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer, 2015.
12. Docker Inc. Docker. https://www.docker.com/. Last retrieved Jan 2016.
13. N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg. Managing multi-cloud systems with CloudMF. In *NordiCloud*, volume 826, pages 38–45. ACM, 2013.
14. J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, 2012.
15. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *CoRR*, abs/1511.04926, 2015.
16. G. E. Gonçalves, P. T. Endo, M. A. Santos, D. Sadok, J. Kelner, B. Melander, and J. Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *CloudCom*, 2011.
17. HashiCorp. Terraform. https://terraform.io/.
18. J. A. Hewson, P. Anderson, and A. D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, 2012.
19. L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1), 2006.

20. OASIS. Cloud Application Management for Platforms. http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.
21. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.
22. Opscode. Chef. http://www.opscode.com/chef/.
23. Puppet Labs. Marionette collective. http://docs.puppetlabs.com/mcollective/.
24. A. Rendl, T. Guns, P. J. Stuckey, and G. Tack. MiniSearch: A Solver-Independent Meta-Search Language for MiniZinc. In *CP*, volume 9255 of *LNCS*, pages 376–392. Springer, 2015.

Appendix B

ABS-YARN: A Formal Framework for Modeling Hadoop YARN Clusters

ABS-YARN: A Formal Framework for Modeling Hadoop YARN Clusters [⋆]

Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, Ming-Chang Lee

Department of Informatics, University of Oslo, Norway
{kellylin,ingridcy,einarj,mclee}@ifi.uio.no

Abstract. In cloud computing, software which does not flexibly adapt to deployment decisions either wastes operational resources or requires reengineering, both of which may significantly increase costs. However, this could be avoided by analyzing deployment decisions already during the design phase of the software development. Real-Time ABS is a formal language for executable modeling of deployed virtualized software. Using Real-Time ABS, this paper develops a generic framework called ABS-YARN for YARN, which is the next generation of the Hadoop cloud computing platform with a state-of-the-art resource negotiator. We show how ABS-YARN can be used for prototyping YARN and for modeling job execution, allowing users to rapidly make deployment decisions at the modeling level and reduce unnecessary costs. To validate the modeling framework, we show strong correlations between our model-based analyses and a real YARN cluster in different scenarios with benchmarks.

1 Introduction

Cloud computing changes the traditional business model of IT enterprises by offering on-demand delivery of IT resources and applications over the Internet with pay-as-you-go pricing [6]. The cloud infrastructure on which software is deployed can be configured to the needs of that software. However, software which does not flexibly adapt to deployment decisions either require wasteful resource over-provisioning or time-consuming reengineering, which may substantially increase costs in both cases. Shifting deployment decisions from the deployment phase to the design phase of a software development process can significantly reduce such costs by performing model-based validation of the chosen decisions during the software design [14]. However, virtualized computing poses new and interesting challenges for formal methods because we need to express deployment decisions in formal models of distributed software and analyze the non-functional consequences of these deployment decisions at the modeling level.

A popular example of cloud infrastructure used in industry is Hadoop [5], an open-source software framework available in cloud environments from vendors

[⋆] Supported by the EU projects H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems* (http://www.hyvar-project.eu) and FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu).

such as Amazon, HP, IBM, Microsoft, and Rackspace. YARN [27] is the next generation of Hadoop with a state-of-the-art resource negotiator. This paper presents ABS-YARN, a generic framework for modeling YARN infrastructure and job execution. Using ABS-YARN, modelers can easily prototype a YARN cluster and evaluate deployment decisions at the modeling level, including the size of clusters and the resource requirements for containers depending on the jobs to be executed and their arrival patterns. Using ABS-YARN, designers can focus on developing better software to exploit YARN in a cost-efficient way.

ABS-YARN is defined using Real-Time ABS, a formal language for the executable modeling of deployed virtualized software [10]. The basic approach to modeling resource management for cloud computing in Real-Time ABS is a separation of concerns between the resource costs of the execution and the resource provisioning at (virtual) locations [18]. Real-Time ABS has previously been used to model and analyze the management of virtual resources in industry [3] and compared to (informal) simulation tools [17]. Although Real-Time ABS provides a range of formal analysis techniques (e.g., [2,30]), our focus here is on obtaining results based on easy-to-use rapid prototyping, using the executable semantics of Real-Time ABS, defined in Maude [12], as a simulation tool for ABS-YARN.

To evaluate the modeling framework, we comprehensively compare the results of model-based analyses using ABS-YARN with the performance of a real YARN cluster by using several Hadoop benchmarks to create a hybrid workload and designing two scenarios in which the job inter-arrival time of the workload follows a uniform distribution and an exponential distribution, respectively. The results demonstrate that ABS-YARN models the real YARN cluster accurately in the uniform scenario. In the exponential scenario, ABS-YARN performs less well but it still provides a good approximation of the real YARN cluster.

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

1. We introduce ABS-YARN, a generic framework for modeling software targeting YARN. Using Real-Time ABS, designers can develop software for YARN on top of the ABS-YARN framework and evaluate the performance of the software model before the software is realized and deployed on a real YARN cluster.
2. ABS-YARN supports dynamic and realistic job modeling and simulation. Users can define the number of jobs, the number of the tasks per job, task cost, job inter-arrival patterns, cluster scale, cluster capacity, and the resource requirement for containers to rapidly evaluate deployment decisions with the minimum costs.
3. We comprehensively evaluate and validate ABS-YARN under several performance metrics. The results demonstrate that ABS-YARN provides a satisfiable modeling to reflect the behaviors of real YARN clusters.

Paper organization. Section 2 provides a background introduction to Real-Time ABS and YARN. Section 3 presents the details of the ABS-YARN framework. In Section 4, we validate ABS-YARN and compare it with a real YARN cluster. Section 5 surveys related work and Section 6 concludes the paper.

Syntactic categories.	*Definitions.*
T in `GroundType`	$P ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\}$
x in `Variable`	$IF ::= \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \}$
s in `Stmt`	$CL ::= \textbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{T}\ \overline{x};]\ \overline{M}\}$
a in `Annotation`	$Sg ::= T\ m\ ([\overline{T}\ \overline{x}])$
g in `Guard`	$M ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\ \}$
e in `Expression`	$a ::= \texttt{Deadline:}\ e \mid DC : e \mid Cost : e \mid a, a$
	$g ::= b \mid x? \mid g \wedge g$
	$s ::= s; s \mid \textbf{skip} \mid \textbf{if}\ b\ \{s\}\ \textbf{else}\ \{s\} \mid \textbf{while}\ b\ \{s\} \mid \textbf{return}\ e$
	$\mid \textbf{duration}(e, e) \mid \textbf{suspend} \mid \textbf{await}\ g \mid \texttt{[}a\texttt{]}\ s \mid x = rhs$
	$rhs ::= e \mid cm \mid \textbf{new}\ C\ (\overline{e}) \mid \textbf{new}\ DeploymentComponent\ (e, e)$
	$cm ::= [e]!m(\overline{e}) \mid x.\textbf{get}$

Fig. 1. Syntax for the imperative layer of Real-Time ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] denote optional elements.

2 Background

2.1 Modeling Deployed Systems Using Real-Time ABS

Real-Time ABS [10] is a formal, executable, object-oriented language for modeling distributed systems by means of concurrent object groups [16], akin to concurrent objects [11], Actors [1], and Erlang processes [7]. Concurrent objects groups execute in parallel and communicate by asynchronous method calls and futures. In a group, at most one process is active at any time, and a queue of suspended processes wait to execute on an object of the group. Processes, which stem from methods calls, are cooperatively scheduled, so active and reactive behaviors can be easily combined in the concurrent object groups. Real-Time ABS combines functional and imperative programming styles with a Java-like syntax and a formal semantics. Internal computations in an object are captured in a simple functional language based on user-defined algebraic data types and functions. A modeler may abstract from many details of the low-level imperative implementations of data structures, but maintain an overall object-oriented design. The semantics of Real-Time ABS is specified in rewriting logic [12], and a model written in Real-Time ABS can be automatically translated into Maude code and executed by the Maude tool.

The imperative layer of Real-Time ABS addresses concurrency, communication, and synchronization based on objects. The syntax is shown in Figure 1. A program P consists of interfaces IF, classes CL with method definitions M, and a main block $\{[\overline{T}\ \overline{x};]\ s\}$. Our discussion focuses on interesting imperative language features, so we omit the explanations of standard syntax and the functional layer (see [16]).

In Real-Time ABS, communication and synchronization are decoupled. Communication is based on asynchronous method calls $f = o!m(\overline{e})$ where f is a future variable, o an object expression, m a method name, and $\overline{e}$ the parameter values for the method invocation. After calling $f = o!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Synchronization is controlled by operations on futures. The statement **await** $f?$ releases the processor while waiting for a reply, allowing other processes to execute. When the reply arrives, the suspended process becomes enabled and the execution may resume. The return value is retrieved by the expression f.**get**, which blocks all execution in the object until the return value is available. The syntactic sugar $x =$ **await** $o!m(\overline{e})$ encodes the standard pattern $f = o!m(\overline{e})$; **await** $f?; x = f$.**get**.

In Real-Time ABS, the timed behavior of concurrent objects is captured by a *maximal progress* semantics. The execution time can be specified directly with *duration* statements, or be implicit in terms of observations on the executing model. Method calls have associated deadlines, specified by `deadline` annotations. The statement **duration**(e_1, e_2) will cause time to advance between a best case e_1 and a worst case e_2 execution time. Whereas duration-statements advance time at any location, Real-Time ABS also allows a separation of concerns between the *resource cost* of executing a task and the *resource capacity* of the location where the task executes. Cost annotations `[Cost: e]` are used to associate resource consumption with statements in Real-Time ABS models.

Real-Time ABS uses *deployment components* to capture the execution capacity of a location in the deployment architecture, on which a number of concurrent objects can be deployed [18]. Each deployment component has its own execution capacity, which will determine the performance of objects executing on the deployment component. Deployment components are dynamically created by $x =$ **new** *DeploymentComponent* (*descriptor*, *capacity*), where x is typed by the DC interface, *descriptor* is a descriptor for the purpose of monitoring, and *capacity* specifies the initial CPU capacity of the deployment component. Objects are deployed on a deployment component using the `DC` annotation on the object creation statement.

2.2 YARN: Yet Another Resource Negotiator

YARN [27] is an open-source software framework supported by Apache for distributed processing and storage of high data volumes. It inherits the advantages of its well-known predecessor Hadoop [5], including resource allocation, code distribution, distributed data processing, data replication, and fault tolerance. YARN further improves Hadoop's limitations in terms of scalability, serviceability, multi-tenancy support, cluster utilization, and reliability.

YARN supports the execution of different types of jobs, including MapReduce, graph, and streaming. Each job is divided into tasks which are executed in parallel on a cluster of machines. The key components of YARN are as follows:

– *ResourceManager* (RM): RM allocates resources to various competing jobs and applications in a cluster, replacing Hadoop's JobTracker. Unlike JobTracker, the scheduling provided by RM is job level, rather than task level. Thus, RM does not monitor each task's progress or restart any failed task. Currently, the default job scheduling policy of RM is CapacityScheduler [23], which allows cluster administrators to create hierarchical queues for multiple

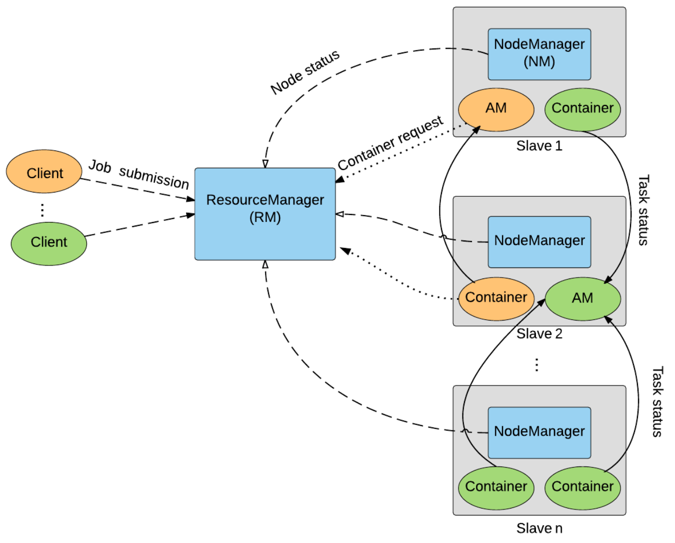


Fig. 2. The architecture of a YARN cluster.

tenants to share a large cluster while giving each tenant a capacity guarantee. The jobs in each queue are scheduled based on a First-in-First-out policy (FIFO), i.e., the first job to arrive is first allocated resources.

– *ApplicationMaster* (AM): This is an instance of a framework-specific library class for a particular job. It acts as the head of the job to manage the job's lifecycle, including requesting resources from RM, scheduling the execution of all tasks of the job, monitoring task execution, and re-executing failed tasks.
– *Containers*: Each container is a logical resource collection of a particular node (e.g., 1 CPU and 2GB of RAM). Clients can specify container resource requirements when they submit jobs to RM and run any kind of applications.

Figure 2 shows the architecture of a YARN cluster, which consists of RM and a set of slave nodes providing both computation resources and storage capacity to execute applications and store data, respectively. A slave node has an agent called NodeManager to periodically monitor its local resource usage and report its status to RM. The execution flow of a job on a YARN cluster is as follows:

1. Whenever receiving a job request from a client, RM follows a pre-defined job scheduling algorithm to find a container from an available slave and initiate the AM of the job on the container.
2. Once the AM is initiated, it starts requesting a set of containers from RM based on the client's container resource requirement and the number of tasks of the job. Basically, each task will be run on one container.

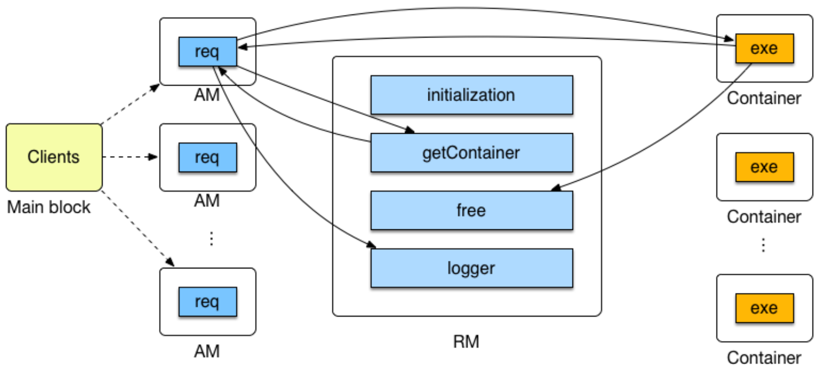


Fig. 3. The structure of the ABS-YARN framework.

3. When RM receives a container request from the AM, it inserts the request into its queue and follows its job scheduling algorithm to allocate a desired container from an available slave node to the AM.
4. Upon receiving the container, the AM executes one task of the job on the container and monitors this task execution. If a task fails due to some errors such as an underlying container/slave node failure, the AM will re-request a container from RM to restart the task.
5. When all tasks of a job finish successfully, implying that the job is complete, the AM notifies the client about the completion.

3 Formal Model of the ABS-YARN Framework

Figure 3 shows the structure of ABS-YARN with classes RM, AM, and Container reflecting the main components of a YARN cluster. In our framework, RM is deployed as an independent deployment component with its own CPU capacity. To model the most general case, we assume that RM has a single queue for all job requests, implying that all jobs are served in a FIFO order. When a client submits a job, an AM object is created for this job, and its **req** method starts requesting containers from RM by invoking the **getContainer** method. If a slave has sufficient resources, a container will be created and returned to the AM. Then the AM submits one task of the job to the allocated container by invoking the **exe** method. When the task terminates, the result is returned to the associated AM, the **free** method is invoked to release the container, and the **logger** method is used to record execution statistics.

ABS-YARN allows modelers to freely determine the scale and resource capacity of a YARN cluster, including (1) the number of slave nodes in the cluster, (2) the CPU cores of each slave node, and (3) the memory capacity of each slave node. To support dynamic and realistic modeling of job execution, ABS-YARN also allows modelers to define the following parameters:

– Number of clients submitting jobs
– Number of jobs submitted by each client
– Number of tasks per job
– Cost annotation for each task
– CPU and memory requirements for each container
– Job inter-arrival pattern. Modelers can determine any kind of job inter-arrival distributions in ABS-YARN.

MapReduce jobs are the most common jobs in YARN, so we focus on modeling their execution in this paper. Each MapReduce job has a map phase followed by a reduce phase. In the map phase, all map tasks are executed in parallel. When all the map tasks have completed, the reduce tasks are executed (normally, each jobs has only one reduce task). The job is completed when all the map and reduce tasks have finished.

The execution time of a task in a real YARN cluster might be influenced by many factors, e.g., the size of the processed data and the computational complexity of the task. To reduce the complexity of modeling the task execution time, ABS-YARN adopts the cost annotation functionality of Real-Time ABS to associate cost to the execution of a task. Hence, the task execution time will be the cost divided by the CPU capacity of the container that executes the task.

In the following, we limit our code presentation to the main building blocks and functionalities to simplify the description.

3.1 Modeling ResourceManager (RM)

The ResourceManager implements the `RM` interface:

```
interface RM {
  Bool initialization(Int s, Int sc, Int sm);
  Pair<Int, Container> getContainer(Int c, Int m);
  Unit free(Int slaveID, Int c, Int m);
  Unit logger(...);}
```

Method `initialization` initializes the entire cluster environment, including RM and `s` slaves. Each slave is modeled as a record in a database `SlaveDB`, with a unique `SlaveID`, `sc` CPU cores, and amount `sm` of memory capacity. After the initialization, the cluster can start serving client requests. Method `getContainer` allows an AM to obtain containers from RM. The size of the required container core and container memory are given by `c` and `m`, respectively. Method `free` is used to release container resources whenever a container finishes executing a task, and method `logger` is used to record job execution statistics, including job ID and job execution time.

The `getContainer` method, invoked by an AM, tries to allocate a container with `c` CPU cores and `m` amount of memory capacity from an available slave to the AM. Each container request is allowed at most `thd` attempts. Hence, as long as `Find==False` and `attempt<=thd` (line 3), the `getContainer` method will keep trying to obtain the database token to ensure a safe database access. The built-in function `lookupDefault` checks each slave in `slaveDB` to find a slave with sufficient resources. If such a slave exists (line 11), the corresponding container will be created as a deployment component with c cores, and the slave's resources will be reduced and updated accordingly (lines 12–14). The successfully generated container is returned to the AM.

However, if no slaves have enough resources, the process will suspend (line 21), allowing RM to process other method activations. The suspended process will periodically check whether any slaves can satisfy the request. If the desired container cannot be allocated within thd attempts, the method terminates and RM is unable to provide the desired container to the AM.

```
 1  Pair <Int, Container> getContainer (Int c, Int m) {
 2    Bool find=False; Int slaveID=1; Int attempt=1;
 3    while (find==False && attempt<=thd){
 4      await dbToken==True;
 5      dbToken==False;
 6      Int i=1;
 7      while (find==False && i<=size(keys(slaveDB))){
 8       Pair<Int,Int> slave= lookupDefault(slaveDB, i, Pair(1,1));
 9       Int free_core= fst(slave);
10       Int free_mem= snd(slave);
11       if (free_core>=c && free_mem >= m){
12         slaveDB=put(slaveDB, i, Pair(free_core-c, free_mem-m));
13         DC s=new DeploymentComponent("slave", map[Pair(CPU,c)]);
14         [DC: s] Container container = new Container(this);
15         find=True;
16         slaveID=i;
17       }
18       i++;
19      }
20      ... // Release dbToken
21      await duration(1,1);
22      attempt++;
23    }
24    if (find==False){ container=null;}
25    return Pair(slaveID, container);
26  }
```

3.2 Modeling ApplicationMaster (AM)

An AM implements the AM interface with a req method to acquire a container from RM and then execute a task on the container. For an AM, the total number of times that req is called corresponds to the number of map tasks of a job (e.g., if a job is divided into 10 map tasks, this method will be called 10 times).

```
interface AM {
  Unit req(Int mNum, Int c, Int m, Rat mCost, Rat rCost);}
```

The req method first invokes the getContainer method and sends a container-resource request (i.e., the parameters c and m) to acquire a container from RM. Since the call is asynchronous, the AM is able to request containers for other tasks of jobID while waiting for the response.

```
Unit req(Int mNum, Int c, Int m, Rat mCost, Rat rCost) {
  ...
3   Pair<Int, Container> p= await rm!getContainer(c, m);
4   Int slaveId=fst(p);
5   Container container=snd(p);
6   if (container!=null){
7     Fut<Bool> f = container!exe(slaveID, c, m, mCost);
8     await f?;
9     Bool map_result = f.get;
10    if (map_result==True){
11      returned_map++;
12      if (returned_map==mNum){
13        Bool red_result;
14        ...//Try to request a container and run the reduce task
15        if (red_result==True){
16          logging the job completion;
17        }
18        else{ logging the reducde-task failure;}
19      }
20    }
21    else{ logging the map-task failure;}
22  }
23  else{ logging unsuccessful container request;}
24 }
```

When a container is successfully obtained, a map task with cost `mCost` can be executed on the container (line 7). The process suspends while waiting for the result of the task execution. Each time when `map_result==True`, the `req` method increases the variable `returned_map` by one. When all map tasks of the job have successfully completed (line 12), the AM proceeds with a container request to run the reduce task of the job with cost `rCost`. Only when all map and reduce tasks are completed (line 15), the job is considered completed.

3.3 Modeling Containers

A container implements the `Container` interface:

```
interface Container{
  Bool exe(Int slaveID, Int c, Int m, Rat tcost);}
```

Method `exe` is used to execute a task on a container. The formal parameters of `exe` consist of `slaveID`, CPU capacity `c`, memory capacity `m`, and the task cost `tcost`. Hence, the task execution time is `tcost/c`. When a task terminates, the `free` method of RM is invoked to release the container, implying that the corresponding CPU and memory resources will be returned back to the slave.

```
Bool exe(Int slaveID, Int c, Int m, Rat tcost){
  [Cost: tcost] ... //executing a task;
  rm!free(slaveID, c, m);
  return true;}
```

4 Performance Evaluation and Validation

To compare the simulation results of ABS-YARN against YARN, we established a real YARN cluster using Hadoop 2.2.0 [5] with one virtual machine acting as

RM and 30 virtual machines as slaves. Each virtual machine runs Ubuntu 12.04 with 2 virtual cores of Intel Xeon E5-2620 2GHz CPU and 2 GB of memory. To achieve a fair validation, we also created an ABS-YARN cluster with 30 slaves; each with 2 CPU cores and 2 GB of memory. To realistically compare job execution performance between ABS-YARN and YARN clusters, we used the following five benchmarks from YARN [23]: **WordCount**, which counts the occurrence of each word in data files; **WordMean**, which calculates the average length of the words in data files; **WordStandardDeviation (WordSD)**, which counts the standard deviation of the length of the words in data files; **GrepSort**, which sorts data files; and **GrepSearch**, which searches for a pattern in data files.

We created a hybrid workload consisting of 22 WordCount jobs, 22 WordMean jobs, 20 WordSD jobs, 16 GrepSort jobs, and 20 GrepSearch jobs. The submission orders of all jobs were randomly determined. Each job processes 1 GB of enwiki data [13] with 128 MB block size (the default block size of YARN [23]). Hence, each job was divided into 8 (=1GB/128MB) map tasks and one reduce task, implying that 9 containers are required to execute each job. We assume that the resource requirement for each container is 1 CPU core and 1 GB RAM for both the ABS-YARN and YARN clusters.

We considered two job inter-arrival patterns in our experiments: Uniform and exponential distribution [20]. In the former, the inter-arrival time between two consecutive jobs submitted by clients are equal. In the latter, job inter-arrival time follows a Poisson process [20], i.e., job submissions occur continuously and independently at a constant average rate. Reiss et al. [25] show that job arrival patterns in a Google trace approximates an exponential distribution. This distribution has also been widely used as job arrival pattern in the literature (e.g., [22, 24]). Based on these distributions, two scenarios were designed:

– *Uniform scenario:* The job inter-arrival time of the workload is 150 sec in the real YARN cluster. In ABS-YARN, this is normalized into 2 time units.
– *Exponential scenario:* The job inter-arrival time of the workload follows an exponential distribution with the average inter-arrival time of 158 sec and a standard deviation of 153 sec in the real YARN cluster. This is normalized into the average inter-arrival time of 158/75 time units and a standard deviation of 153/75 time units in the ABS-YARN cluster.

The following metrics were used to evaluate how well ABS-YARN can simulate job scheduling, job execution behavior, and job throughput of YARN:

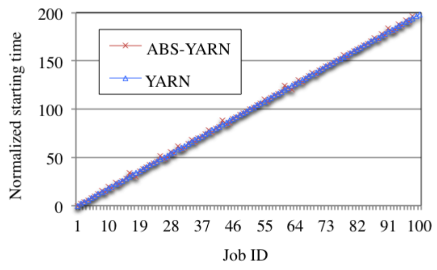
– Starting time of all jobs of the workload
– Finish time of all jobs of the workload
– The number of cumulative completed jobs
– Total number of completed jobs

4.1 Validation Results in the Uniform Scenario

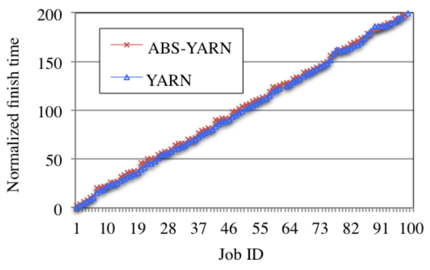
In order to achieve a fair comparison, we conducted the uniform scenario on the YARN cluster to obtain the average map-task execution time (AMT) and

Table 1. The average map-task execution time (AMT), average reduce-task execution time (ART), normalized map-task cost annotation (MCA), and normalized reduce-task cost annotation (RCA) in the uniform scenario.

Benchmark	AMT (sec)	ART (sec)	MCA	RCA
WordCount	162.64	251.01	2.17 (=162.64/75)	3.35 (251.01/75)
WordMean	107.10	139.94	1.43 (=107.10/75)	1.87 (=139.94/75)
WordSD	108.23	162.27	1.44 (=108.23/75)	2.16 (=162.27/75)
GrepSort	20.39	38.44	0.27 (=20.39/75)	0.51 (=38.44/75)
GrepSearch	31.22	55.97	0.42 (=31.22/75)	0.75 (=55.97/75)



(a) The normalized starting time

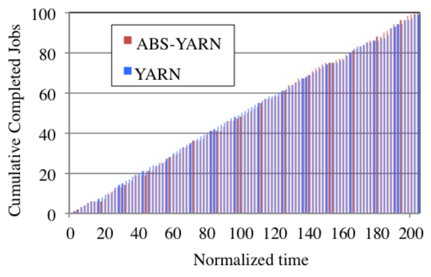


(b) The normalized finish time

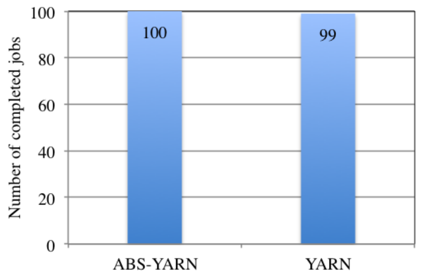
Fig. 4. The normalized time points of all jobs in the uniform scenario.

average reduce-task execution time (ART) for each job type. The results are listed in Table 1. After that, we respectively normalized each AMT and ART into a map-task cost and a reduce-task cost for ABS-YARN by dividing the AMT value by 75 and dividing the ART value by 75 (Note that 75 is half of the job inter-arrival time for the uniform scenario). With the corresponding map-task cost annotation (MCA) and reduce-task cost annotation (RCA), we simulated the uniform scenario on ABS-YARN.

Figure 4(a) shows the normalized starting time of all jobs in both clusters. We can see that the two curves are almost overlapping. The average time difference between ABS-YARN and YARN is 0.02 time units with a standard deviation of 1.73 time units, showing that ABS-YARN is able to precisely capture the job scheduling of YARN in the uniform scenario. Figure 4(b) depicts all job finish time in both clusters. The average difference between ABS-YARN and YARN is 2.67 time units with a standard deviation of 1.81 time units, indicating that the framework can accurately model how containers execute jobs in a real YARN cluster. Based on the results shown in Figure 4, we can derive that the cumulative numbers of completed jobs between the two clusters are close (see Figure 5(a)). The average error is approximately 2.52%, implying that ABS-YARN can precisely reflect the operation of YARN in the uniform scenario. Figure 5(b) shows that 100 jobs of the workload successfully finished in the ABS-YARN cluster, but 99 jobs of the workload completed in the YARN cluster



(a) Cumulative completed jobs



(b) Total number of completed jobs

Fig. 5. The cumulative completed jobs and the total number of completed jobs in the uniform scenario.

since the remaining one job could not obtain sufficient containers to execute its tasks. The job completion error of ABS-YARN is only 1.01%. Based on the above-mentioned results, it is evident that the ABS-YARN framework offers a superior modeling of YARN in the uniform scenario.

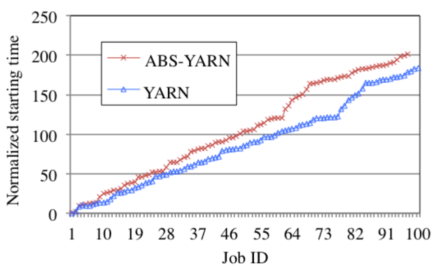
4.2 Validation Results in the Exponential Scenario

In this section, we compare ABS-YARN and YARN under the exponential scenario. Similar to the uniform scenario, we performed a normalization by executing the exponential scenario on the YARN cluster to derive a map-task cost annotation and a reduce-task cost annotation for each job type. The results are listed in Table 2. Note that regardless of which job type was tested, the corresponding average map-task and reduce-task execution time were apparently higher than those in the uniform scenario. The main reason is that the job inter-arrival time in the exponential scenario had a much higher standard deviation, implying that many jobs might compete for containers at the same time. However, due to the limited container resources, these jobs had to wait for available containers and hence prolonged their execution time.

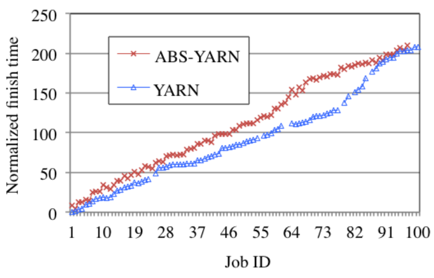
Table 2. The AMT, ART, MCA, and RCA in the exponential scenario.

Benchmark	AMT (sec)	ART (sec)	MCA	RCA
WordCount	295.47	430.24	3.94 (=295.27/75)	5.74 (430.24/75)
WordMean	139.98	201.11	1.87 (=139.98/75)	2.68 (=201.11/75)
WordSD	238.46	312.38	3.18 (=238.46/75)	4.17 (=312.38/75)
GrepSort	37.38	62.06	0.50 (=37.38/75)	0.83 (=62.06/75)
GrepSearch	173.92	205.94	2.32 (=173.92/75)	2.75 (205.94/75)

The normalized job starting time illustrated in Figure 6(a) show that the ABS-YARN cluster follows the same trend as the YARN cluster. However, as more jobs were submitted, their starting time in ABS-YARN were later than



(a) The normalized starting time



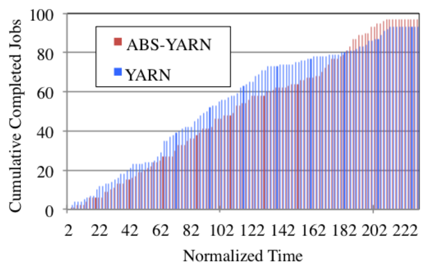
(b) The normalized finish time

Fig. 6. The time points of all jobs in the exponential scenario.

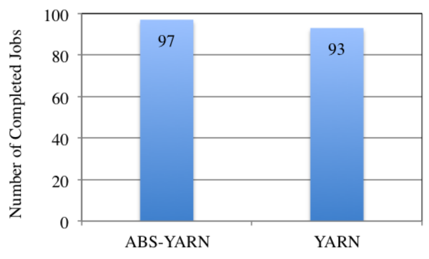
those in the YARN cluster. The average time difference is around 19.48 with standard deviation of 12.92. The key reasons are two. First, the normalized map-task (reduce-task) cost annotations used by ABS-YARN were based on average map-task (reduce-task) execution time of the entire workload, which were longer than the actual map-task (reduce-task) execution time spent by the real YARN cluster in the early phase of the workload execution. Second, the number of available containers gradually decreased when more jobs were submitted to the ABS-YARN cluster. For these two reasons, the starting time of the subsequent jobs were delayed.

Figure 6(b) depicts the normalized job finish time of the two clusters under the exponential scenario. We can see that during the workload execution, many jobs in the ABS-YARN cluster finished later than the corresponding jobs in the YARN cluster. The reasons are the same, i.e., the map-task (reduce-task) cost annotation values were derived from the corresponding average map-task (reduce-task) execution time, which were usually higher than the actual execution time in the YARN cluster during the early stage of the workload. Nevertheless, the results show that even under a heavy and dynamic workload, the ABS-YARN framework can still adequately model YARN.

The cumulative number of completed jobs illustrated in Figure 7(a) shows that during most of the workload execution, the ABS-YARN cluster finished fewer jobs than the YARN cluster for the above mentioned reasons. However, in the late stage, the ABS-YARN cluster had more completed jobs than the YARN cluster. This phenomenon can also be deduced from Figure 6 since seven jobs could not complete by the YARN cluster. The average difference of the cumulative workload completion between ABS-YARN and YARN is 14.49%. Due to failing to get containers, 97 jobs and 93 jobs (as shown in Figure 7(b)) were finished by the ABS-YARN cluster and the YARN cluster, respectively. Although the job completion error of ABS-YARN is increased to 4.3% from the uniform scenario to the exponential scenario, the above results still demonstrate that the ABS-YARN framework provides a satisfiable modeling for YARN.



(a) Cumulative completed jobs



(b) Total number of completed jobs

Fig. 7. The cumulative completed jobs and the total number of completed jobs in the exponential scenario.

5 Related Work

General-purpose modeling languages provide abstractions where the main focus has been on describing functional behavior and logical composition. However, this is inadequate for virtualized systems such as clouds when the software's deployment influences its behavior and when virtual processors are dynamically created. A large body of work on performance analysis using formal models can be found based on, e.g., process algebra [9], Petri Nets [26], and timed and probabilistic automata [4, 8]. However, these works mainly focus on non-functional aspects of embedded systems without associating capacities with locations. A more closely related technique for modeling deployment can be found in an extension of VDM++ for embedded real-time systems [28], in which static architectures are explicitly modeled using buses and CPUs with fixed resources.

Compared to these languages, Real-time ABS [10,18] provides a formal basis for modeling not only timed behavior but also dynamically created resource-constrained deployment architectures, which enables users to model feature-rich object-oriented distributed systems with explicit resource management at an abstract yet precise level. Case studies validating the formalization proposed in Real-Time ABS include Montage [17] and the Fredhopper Replication Server [3]. Both case studies address resource management in clouds by combining simulation techniques and cost analysis. Different from these case studies, this paper uses Real-Time ABS to create a formal framework for YARN and comprehensively compare this framework with a real YARN cluster.

In recent years, many simulation tools have been introduced for Hadoop, including MRPerf, MRSim, and HSim. MRPerf [29] is a MapReduce simulator designed to understand the performance of MapReduce jobs on a specific Hadoop parameter setting, especially the impact of the underlying network topology, data locality, and various failures. MRSim [15] is a discrete event based MapReduce simulator for users to define the topology of a cluster, configure the specification of a MapReduce job, and simulate the execution of the job running on the cluster. HSim [21] models a large number of parameters of Hadoop, including nodes, cluster, and simulator parameters. HSim also allows users to describe their own job specification. All the above-mentioned simulators target Hadoop rather than YARN. Due to the fundamental difference between Hadoop and YARN, these simulators are unable to simulate YARN. Besides, these simulators concentrate on simulating the execution of a single MapReduce job and compare the corresponding simulation results with the actual results on real Hadoop systems. However, this is insufficient to confirm that they can faithfully simulate Hadoop when multiple jobs are running on Hadoop. Similar work can also be found in [19]. Different from all these simulators, the proposed ABS-YARN framework is designed to model a set of jobs running on YARN, rather than just one job. With ABS-YARN, users can comprehend the performance of YARN under a dynamic workload.

To our knowledge, the Yarn Scheduler Load Simulator (SLS) [31] is the only simulator currently designed for YARN, but it concentrates on simulating job scheduling in a YARN cluster. Besides, SLS does not provide any performance evaluation to validate its simulation accuracy. Compared with SLS, ABS-YARN provides a formal executable YARN environment. In this paper, we also present a comprehensive validation to demonstrate its applicability.

6 Conclusion and Future Work

This paper has presented the ABS-YARN framework based on the formal modeling language Real-Time ABS. ABS-YARN provides a generic model of YARN by capturing the key components of a YARN cluster in an abstract but precise way. With ABS-YARN, modelers can flexibly configure a YARN cluster, including cluster size and resource capacity, and determine job workload and job inter-arrival patterns to evaluate their deployment decisions.

To increase the applicability of formal methods in the design of virtualized systems, we believe that showing a strong correlation between model behaviors and real system results is of high importance. We validated ABS-YARN through a comprehensive comparison of the model-based analyses with the actual performance of a real YARN cluster. The results demonstrate that ABS-YARN is accurate enough to offer users a dependable framework for making deployment decisions about YARN at design time. In addition, the provided abstractions enable designers to naturally model and design virtual systems at this complexity, such as enhancing YARN with new algorithms.

In future work, we plan to further enhance ABS-YARN by incorporating multi-queue scheduler modeling, slave and container failure modeling, and distributed file-system modeling. Modeling different job types will also be considered. Whereas this paper has focussed on the accuracy of the ABS-YARN framework, our ongoing work on a more powerful simulation and visualization tool for Real-Time ABS will improve the applicability of ABS-YARN.

Acknowledgement. The authors thank NCLab at National Chiao Tung University, Taiwan for providing computation facilities for the YARN cluster used in our experiments.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *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, 2014.
3. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling and analysis 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.
4. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Proc. FORMATS'03*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2003.
5. Apache Hadoop. `http://hadoop.apache.org/`.
6. M. Armbrust, A. Fox, R. Griffith, Anthony D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
7. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
8. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Performance evaluation and model checking join forces. *Commun. ACM*, 53(9):76–85, 2010.
9. F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. Space-aware ambients and processes. *Theoretical Computer Science*, 373(1–2):41–69, 2007.
10. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
11. D. Caromel and L. Henrio. *A Theory of Distributed Objects.* Springer, 2005.
12. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science.* Springer, 2007.
13. enwiki. `http://dumps.wikimedia.org/enwiki/`.
14. R. Hähnle and E. B. Johnsen. Designing resource-aware cloud applications. *IEEE Computer*, 48(6):72–75, 2015.
15. S. Hammoud, M. Li, Y. Liu, N. K. Alham, and Z. Liu. MRSim: A discrete event based MapReduce simulator. In *2010 seventh International Conference on Fuzzy Systems and Knowledge Discovery*, FSKD '10, pages 2993–2997. IEEE, 2010.
16. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
17. E. B. Johnsen, R. Schlatte, and S. L. 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, Nov. 2012.

18. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.
19. W. Kolberg, P. de B. Marcos, J. C. Anjos, A. K. Miyazaki, C. R. Geyer, and L. B. Arantes. MRSG - a MapReduce simulator over SimGrid. *Parallel Computing*, 39(4):233–244, 2013.
20. L. B. Koralov and Y. G. Sinai. *Theory of Probability and Random Processes*. Berling: Springer-Verlag, 2007.
21. Y. Liu, M. Li, N. K. Alham, and S. Hammoud. HSim: a MapReduce simulator in enabling cloud computing. *Future Generation Computer Systems*, 29(1):300–308, 2013.
22. C. Luo, J. Zhan, Z. Jia, L. Wang, G. Lu, L. Zhang, C.-Z. Xu, and N. Sun. Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications. *Frontiers of Computer Science*, 6(4):347–362, 2012.
23. A. Murthy, V. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 2014.
24. B. Palanisamy, A. Singh, L. Liu, and L. Bryan. Cura: A cost-optimized model for MapReduce in a cloud. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1275–1286. IEEE, 2013.
25. C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical Report ISTC-CC-TR-12-101, Intel Science and Technology Center for Cloud Computing, Carnegie Mellon University, Apr. 2012. Available via web: `http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/ISTC-CC-TR-12-101.pdf`.
26. M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proc. Design Automation Conference*, DAC '99, pages 805–810. ACM, 1999.
27. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In G. M. Lohman, editor, *ACM Symposium on Cloud Computing (SOCC'13)*, pages 5:1–5:16, 2013.
28. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In J. Misra, T. Nipkow, and E. 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.
29. G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *IEEE international Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '09, pages 1–11. IEEE, 2009.
30. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.
31. Yarn Scheduler Load Simulator (SLS). `https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html`.