Declarative Elasticity in ABS

Stijn De Gouw, Jacopo Mauro, Behrooz Nobakht, Gianluigi Zavattaro

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 Traditional development methodologies that separate software design from application deployment have been replaced by approaches such as continuous delivery or DevOps, according to which deployment issues should be taken into account already at the early stages of development.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 modeling 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 incremental 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 integration of deployment in software modeling 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] 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 incremental 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]). Executable 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 incremental deployment decisions like, e.g., the need to upscale or downscale the modeled system at run-time 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 classes exposing the methods `deploy` and `undeploy` to 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 incremental 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 by 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. Section 7 discuss the related literature while in Section 8 we draw some concluding remarks.

2   The Fredhopper Cloud Services

Fredhopper provides 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. Software services are deployed as *service instances*. Each instance offers the same service and is exposed via Load Balancer endpoints that distribute requests over the service instances.

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-users are plays an important role (typical lows in demand are observed between 2 am and 5 am). Figure 1 shows a real-world graph for a single day (with data up to 18:00) plotting 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). The 2a - 5am low is clearly visible.
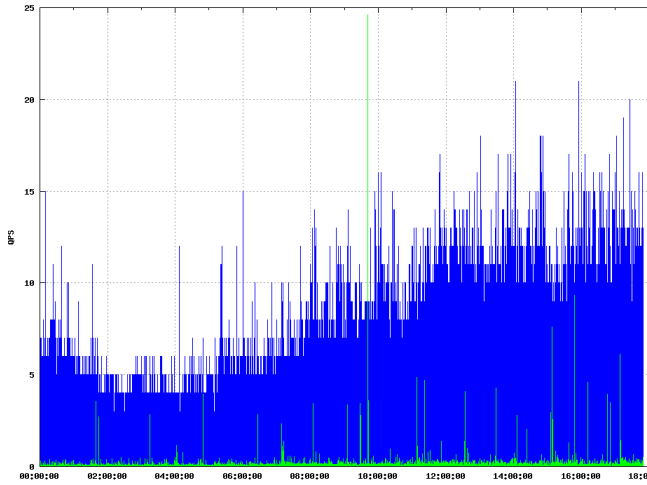
Fig. 1. Number of queries per second (in green the query processing time).

Peaks typically 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 (using 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 model of the Fredhopper Cloud Services, enables the modeling of automatic 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.

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.

4

- 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 programmer 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 provider "Amazon" is modeled as the object `cProv` of type `CloudProvider`. The fact that "Amazon" can provide a virtual machine of type "c3" is modeled by calling `addInstanceDescription` in Line 2. With this instruction we also specify that c3 virtual machines cost 0,210 cents per hour, 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 on the deployment component `dc`.

ABS supports declaring interface hierarchies and defining 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 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 the cost and 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 above two annotations, to be included before the declaration of the class `QueryServiceImpl` in the above ABS code, describe two possible deployment scenarios for objects of that class. The first annotation models the deployment of a Query Service in staging mode, the second one models the deployment in live mode. A Query Service in staging mode requires 2 cores and 7GB of RAM. In live mode, 1 core and 3GB of RAM suffices. Creating a Query Service object requires the instantiation of its two initialization parameters `ds` and `staging`. The second parameter should be instantiated with `True` or `False` depending on the deployment scenario. The first parameter is required (keyword `Req` in the annotation): this means that the Query Service requires a reference to an object of type `DeploymentService` passed via the `ds` initialization parameter.

4 The Declarative Requirement Language DRL

Computing a deployment configuration requires taking into account the expectations of the ABS programmer. For example, in the Fredhopper Cloud Services, one initial goal is to deploy with reasonable cost a given number of Query Services and a Platform Service, possibly located on different machines to improve fault tolerance, and later on to upscale (or subsequently downscale) the system according to the monitored traffic. Each desiderata can be expressed with a corresponding expression in *Declarative Requirement Language* (DRL): a new language for stating constraints a configuration to be computed should satisfy.

```
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 '[' ID ']' | 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 1. DRL grammar.

As shown in Table 1, that reports an excerpt of the DRL grammar,[5] a desiderata is a (possibly quantified) Boolean formula `b_expr` obtained by using the usual logical connectives over comparisons between arithmetic expressions. An atomic arithmetic expression is an integer (Line 6), a sum statement (Line 8) or an identifier for the number of deployed objects (Line 9). The number of objects to deploy using a given scenario is defined by its class name and the scenario name enclosed in square brackets (Line 12). For example, the below formula requires deploying at least one object of class `QueryServiceImpl` in staging mode.

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

The square brackets are optional (Line 12 - first option) for objects with only one default deployment scenario. Regular expressions (`RE` in Line 12) can match objects deployed using different scenarios. The number of deployed objects can 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
```

---

[5] The complete grammar defined using the ANTLR compiler generator is available at https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/decl_spec_lang/DeclSpecLanguage.g4.

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 13, variables in quantifiers and sums can range over all the objects (`'obj'`), all the deployment components (`'DC'`), or just 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, the constraint enforcing that every Query Service has a Deployment Service installed on its virtual machine is as follows.

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

Here `impl` stands for logical implication. The regular expression `'.*'` allows us to match with both deployment modalities for the Query Service (`staging` and `live`). Finally, specifying that the load balancer must be installed on a dedicated virtual machine (without other Service instances) can be done as follows.

```
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 realize automatic deployment. The key idea of SmartDepl is to allow the user on the one hand to declaratively specify the desired deployments and, on the other hand, to develop its program abstracting from concrete deployment decisions. More concretely, deployment requirements are specified as program annotations. SmartDepl processes each of these annotations and generates for each of them a new class that specifies the deployment steps to reach the desired target. Then this class can be used to trigger the execution of the deployment, and 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

---

[6] We assume that every deployment desiderata expressed in DRL deals with only a bounded number of deployment components (the bound is a configuration parameter for SmartDepl). Notice that this does not mean that the total number of deployment components in an application is bound, as the deployment can be repeated an unbounded number of times.

[7] DRL improves on the specification language presented in [10] because the addition of the quantifiers and sums allow to write the desiderata more concise and naturally.

```
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. An example of a deployment annotation.

user wants to add a Query Service instance in live mode. The annotation that
describes this requirement is the JSON object defined in Table 2.[8]

In Line 1, the keyword `"id"` specifies that the name of the class with the de-
ployment code, to be synthesized by SmartDepl, is `AddQueryDeployer`. As we will
see later, this class exposes methods to be invoked to actually execute deploy-
ment actions that modifies the current deployment according to the requirements
in the deployment annotation. The second line contains the declarative specifica-
tion of the desired configuration in DRL. Deploying a new instance of the Query
Service may involve other relevant objects from the surrounding environment,
such as the `PlatformService` or a `LoadBalancerService`. Which objects are rele-
vant may come from business, security or performance reasons, thus in general
it may be undesirable to select or create automatically a Service instance of the
right type. SmartDepl is flexible in this regard: the user supplies the appropriate
ones. By using the keyword `"obj"`, Lines 3-18 list the appropriate objects. Since
these object are already available, they need not be deployed again. The names
of these objects are specified with the keyword `"name"` (Lines 3,9,14), the pro-
vided interfaces with the keyword `"port"` (Lines 5-6,11,16) with the amount of
services that can use it (keyword `"num"` in Lines 7,12,17 — in this case a -1 value

---

[8] To facilitate the interoperability between ABS and SmartDepl we have
adopted a JSON syntax for the deployment annotations. For the inter-
ested reader the formal specification of the JSON annotations is defined
in https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/spec/
smart_deploy_annotation_schema.json.

means that the object can be used by an unbounded number of other objects), and the object interface with keyword `"interface"` (Lines 8,13,18). Finally, with the keyword `"DC"`, the user specifies if there are existing deployment components with free resources that can be used to deploy new objects. In this case, for fault tolerance reasons the user wants to deploy the Query Service in a new machine and therefore the `"DC"` is empty (Line 19).

Once the annotation is given, the user may freely use this class. For instance, the below ABS code scales the system up or down based on a monitor decision.

```
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(); } }
```

Every time an upscale is needed, an object of class AddQueryDeployer (the name associated with the annotation previously discussed) is created. The idea is to store the references to these deployment objects in a list called depObjList. We now discuss the initialization parameters for such objects. The first parameter is the cloud provider, as defined for instance in Section 3. The next parameters are the objects already available for the deployment that do not need to be re-deployed. These are given according to the order they are defined in the annotation in Table 2. The generated class implements the SmartDeployInterface 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 deploy method, iii) getter methods to retrieve the list of new objects and deployment components created by running the deploy method (e.g., a call depObj.getIQueryService() retrieves the list of all the Query Services created by depObj.deploy()). The actual addition of the Query Service is performed in Line 5 with the call of the deploy method. If the monitor decides to downscale (Line 7), the last deployment solution is retrieved (Line 8), and the corresponding deployment actions are reverted by calling the undeploy method.[9]

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, computes the optimal configuration satisfying the user needs.[10] The cost annotations (see Section 3) are used to compute a configuration

---

[9] 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.

[10] SmartDepl uses Zephyrus2 (freely available at https://jacopomauro@bitbucket.org/jacopomauro/zephyrus2.git) since it allows the use of a new expressive lan-

that satisfies the constraints, minimizes the cost of the deployment components that need to be created and, in case of ties, minimizes the number of created objects. The user is notified if no configuration exists that satisfies the desiderata. Once a configuration is obtained, SmartDepl uses topological sorting to take into account all the object dependencies and computes the sequence of deployment instructions to realise the desirable configuration. SmartDepl exploits Delta Modeling [7] to generate the code of the classes and methods to inject into the interface. SmartDepl also notifies the user when it is unable to generate a sequence of deployment actions due to mutual dependencies between the objects.[11]

As an example the deploy code generated by SmartDepl for the annotation defined in Table 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 }
```

At Line 3, a new deployment component c3_0 is created. In Lines 4-5 an object of class DeploymentService is created, since every Query Service requires a corresponding Deployment Service (it is one of the required parameters, cf. Section 3) to be deployed before the Query Service. In Lines 8-9 the desired object of class IQueryService is created. Both objects are deployed on c3_0.

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

SmartDepl is open source, available at https://github.com/jacopoMauro/abs_deployer/tree/smart_deployer and to increase its portability it can be installed also by using the Docker container technology [12]. As illustrated in Figure 2, SmartDepl has also been integrated into the ABS toolchain,[12] an IDE for a collection of tools for writing, inspecting, checking, and analyzing ABS programs developed within the Envisage European project.

---

guage 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].

[11] This occurs when the creation of an object requires the execution of a complex protocol, such as what happens for the boostrapping of Linux distributions [1].
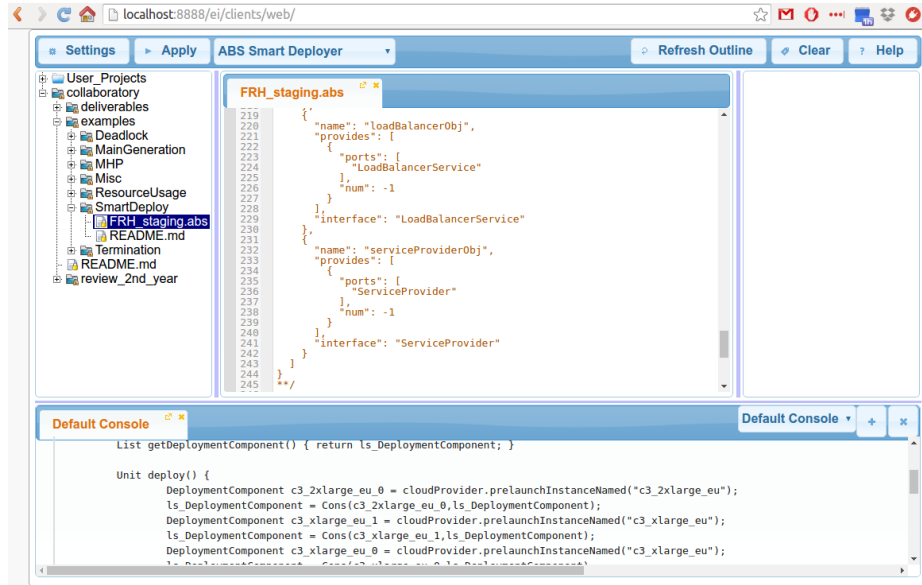
[12] http://abs-models.org/installation/

Fig. 2. SmartDepl execution within the ABS toolchain IDE.

## 6   Application to the Fredhopper use case

In this section we report on 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 extensive profiling of the in-production system, the cost of its services are known.

SmartDepl was used twice: to synthesize the initial static deployment of the entire framework and to add (and later remove) instances of the Query Service if the system needs to scale. Since the Fredhopper Cloud Services uses 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)[13] of Amazon. For fault tolerance and stability, Fredhopper Cloud Services uses instances in multiple regions in Amazon (regions are geographically separate areas, so even if there is a force majeure in one region, other regions may be unaffected). 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" is an American region).

The static deployment of the Fredhopper Cloud Services requires deploying a Load Balancer, a Platform Service, a Service Provider and 2 Query Services with at least one in staging mode. This is expressed as follows.

LoadBalancerServiceImpl $= 1$ and PlatformServiceImpl $= 1$ and

---

[13] https://aws.amazon.com/ec2/instance-types/
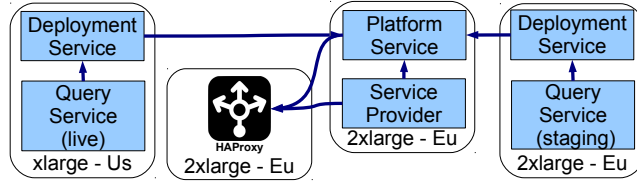
Fig. 3. Example of automatic objects allocation to deployment components.

```
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 is expressed as shown in Section 4. The requirement that a Service Provider is present on every machine containing a Platform Service is expressed by:

```
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 resilience, we require that the Load Balancer, the Query/Deployment Services, and the Platform Service/Service Provider are never co-located on the same virtual machine. The end of Section 4 shows how this is expressed.

To handle catastrophic failures, the Fredhopper Cloud Services aim to balance the Query Services between the regions (see Section 2). This is enforced by constraining the number of the Query Services in the different data centers to be equal. In DRL this is expressed with 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 is expressed by:

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

From this specification SmartDepl computes the initial configuration in Figure 3, which minimizes the total costs per interval. It deploys the Load Balancer, Platform Service and one staging Query Service on three "2xlarge" instances in Europe, and deploys a live Query service on an "xlarge" instance in US.

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 4 and 5 show some of the main metrics for a single customer used to determine the scaling. The timescale in the figures is 1
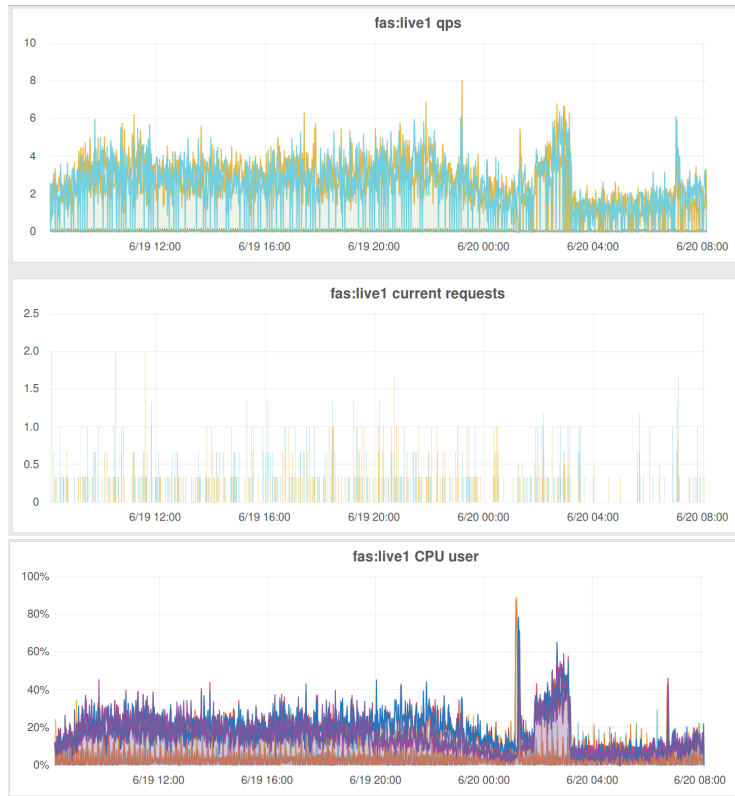
Fig. 4. Metrics graphed over a single day for a customer (a).

day, but this can be adjusted to see trends over longer periods, or zoom in on a short period. The figures show that the number of queries served per second (qps, first graph of Figure 4) is relatively high and the requests (Figure 4, second graph) are fairly low, so requests are not queuing. Furthermore the CPU usage (Figure 4, third graph) and memory consumption with small swap space used (Figure 5, second and third graphs) 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. In contrast, if there is unnecessary overcapacity, the most recent ones can be shut down. Since the Cloud operations team currently manually decides to scale, and Fredhopper has very aggressive SLAs, the 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 leverage the extensive tool-supported analyses available for ABS [3, 11, 15, 25]. For example, by using monitors to track the quality of services, SmartDepl allows to reason on a rigorous basis on the scaling decisions and their impact on the SLA agreed with the customers.
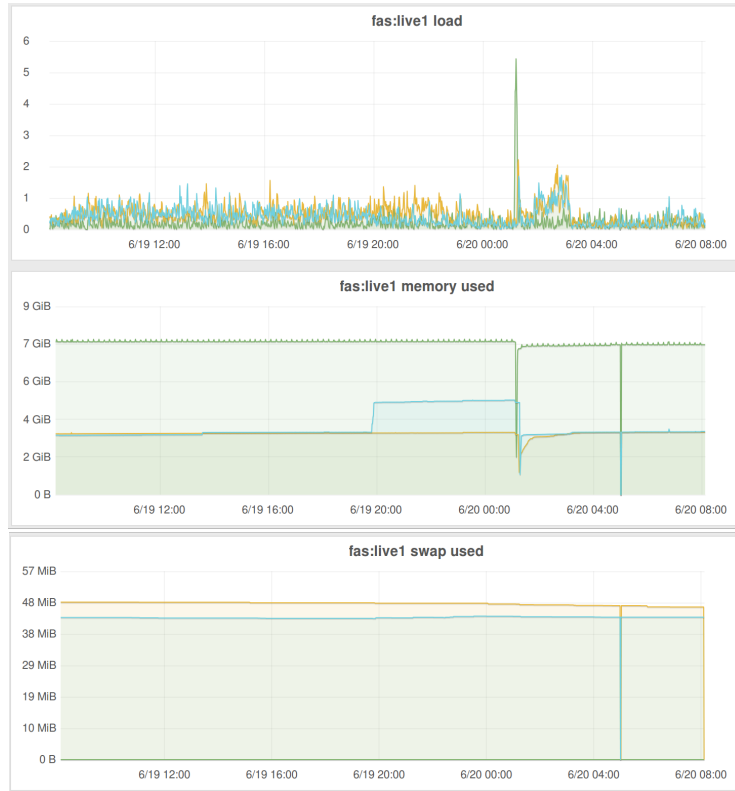
Fig. 5. Metrics graphed over a single day for a customer (b).

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 Cloud API offered by ABS for managing virtual resources. Executable code is automatically generated from ABS for any of the infrastructures for which an implementation of the Cloud API exists (e.g., Amazon, Docker, OpenStack).

To automatically generate the scaling deployment configuration, SmartDepl uses all the previous specifications, 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. The ABS model used with all the annotations and specifications and an example of generated code is available at https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/test/.

15

7 Related Work

Many management tools for bottom-up deployment exist, e.g., CFEngine [6], Puppet [19], MCollective [23], and Chef [22]. 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 them for an optimal resource allocation. The alternative holistic approach allows modeling the entire application and derives the deployment plan top-down. 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], JCloudScale [26], Apache Brooklyn [4], and tools supporting the Cloud Application Management for Platforms protocol [20]. A first attempt to combine the holistic and bottom-up approaches is reported in [5]: 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).

8 Conclusions

We presented an extension of the ABS specification language that supports modeling deployment in a declarative manner: the programmer specifies deployment constraints, and a solver synthesizes ABS classes with methods that execute deployment actions to reach an optimal deployment configuration that satisfies the constraints. Our approach, which is inspired by [9] and significantly improves our initial work [10], can be easily applied to any other object-oriented language that offers primitives for the acquisition and release of computing resources.

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*, 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 *ETAPS*, 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 *ESOCC*, 2015.

6. M. Burgess. A Site Configuration Engine. *Computing Systems*, (2), 1995.

7. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability Modelling in the ABS Language. In *FMCO*, 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*, 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 *ESOCC*, 2015.

11. C. C. Din, R. Bubel, and R. Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In *CADE*, 2015.

12. Docker Inc. Docker. https://www.docker.com/.

13. N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg. Managing multi-cloud systems with CloudMF. In *NordiCloud*, 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*, 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*, (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*, 2015.

25. P. Y. H. Wong, R. Bubel, F. S. de Boer, M. Gómez-Zamalloa, S. de Gouw, R. Hähnle, K. Meinke, and M. A. Sindhu. Testing abstract behavioral specifications. *STTT*, 17(1):107–119, 2015.

26. R. Zabolotnyi, P. Leitner, W. Hummer, and S. Dustdar. JCloudScale: Closing the Gap Between IaaS and PaaS. *ACM Trans. Internet Techn.*, 15(3):10, 2015.