SmartDepl

http://abs-models.org/smartdepl-tutorial/

Jacopo Mauro

ENVISAGE
http://www.envisage-project.eu/

Introduction

In the following we present the SmartDepl tool.

In this tutorial 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, dubbed SmartDepl, 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.

After working through this tutorial, you will know how to annotate classes and use SmartDepl to generate the code to optimally deploy objects within DC.

Contents

1 General Overview

The key idea of SmartDepl is to allow declarative specifications what the user wants to deploy, and develop the program abstracting from concrete deployment decisions. More concretely, the user specifies her deployment requirements as program annotations. SmartDepl processes them and generates for every annotation a new class that specifies the deployment steps to reach the desired target. The user can use this class to trigger the execution of the deployment, and to undo it in case the system needs to downscale.

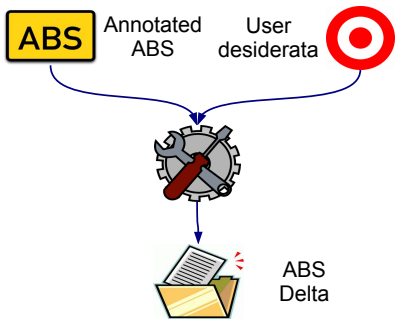

Figure 1: SmartDepl workflow.

As depicted in Figure 1, SmartDepl takes two different inputs: the ABS program annotated with cost annotations and the user desiderata. In Section 2 we present how ABS classes can be annotated to define every objects consumption. In Section 3 we present instead how the user can produce the annotations for the automatic generation of the deployment code. Then, in Section 4 we show how to wrap up all these things and exploit them to generate a configuration.

In particular, we will use the Fredhopper Cloud Services use case as a running example. 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. Figure 2 shows a block diagram of the Fredhopper Cloud Services.

The number of requests can vary greatly over time, and typically depends on several factors. Scaling or downscaling the application 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.

- 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.

Figure 2: The architecture of the Fredhopper Cloud Services

- 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.

We will show how SmartDepl can be used to generate automatically configurations satisfying all these constraints. In the following we assume that the reader is familiar with ABS.

2 Cost annotations

In this section we describe how an existing ABS program can be annotated to related objects with their resource consumption.

Ideally, we would like to have a measure of the resource consumption associated to every object that can be created. In this way we can have a precise estimation of the resources needed by the overall system and take deployment decisions accordingly. SmartDepl exploits the ABS Cloud API that supports a limited number of resources. In particular, SmartDepl is able to take into account the memory consumption, the number of cores, and the number of instruction of the computational unit. In the following, for simplicity, we will focus only on the memory consumption and the number of cores that according to the cloud API are denoted as `Memory` and `Cores`, respectively.

SmartDepl requires an annotation for every relevant class that can be involved in the automatic generation of the deploying code. Intuitively, an annotation for the class `C` describes: (i) the maximal resource consumption of an object `obj` of the class `C`, (ii) the requirements on the initialization parameters for class `C` (for instance, at least two services should be present in the initialization list of a load balancer), and (iii) how many other objects in the deployed system can use the functionalities provided by `obj`.

Examples of cost annotations for the specification of the `QueryServiceImpl` of the Fredhopper Cloud Services) are the following ones.

```
[ 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)] ]
class QueryServiceImpl(DeploymentService ds, Bool staging) implements IQueryService { ... }
```

Annotations are defined before the definition of the class. In this case the 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 mode, the second captures 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 an object of type DeploymentService (stored in the parameter dc). The second parameter, staging, indicates if the instance operates in staging mode.

In general, as can be seen from the grammar of the ABS annotations reported in Table 1, given a class `C`, an annotation `ann` is simply a list of comma separated expressions `expr` where the expressions are of the following types.

- `Name(X)`: associates a name `X` to the annotation. The name, also called *scenario name* or simply scenario, identifies unequivocally the annotation in case of different annotations for the same class `C`, each one representing a different way for deploying objects of that class. default value `Def`. This expression can be left unspecified in at most one of the annotations of a class: in this case the name is set to the default value `Def`.
- `MaxUse(X)`: indicates that an object `obj` of class `C` can be used in the creation of at most `X` other objects. This parameter expresses the constraint that in the specified deployment scenario, `obj` can provide its functionalities only to a limited number of other client objects. By default, if this field is absent, an unlimited number of client objects is considered.

```
1  ann
2    : '[Deploy: scenario[' expr (',' expr)* ']]';
3  expr
4    : 'Name(' STRING ')'
5    | 'MaxUse(' INT ')'
6    | 'Cost(' STRING ',' INT ')'
7    | 'Param(' STRING ',' paramKind ')';
8  paramKind
9    : User
10   | 'Default(' STRING ')'
11   | Req
12   | 'List(' INT ')';
```

Table 1: Grammar of ABS annotations.

- `Cost( r, X )`: indicates that an object `obj` of class `C` consumes at most `X` units of the resource `r`.
- `Param( param, kind )`: indicates how the initialization parameters `param` for class `C` must be instantiated when an object `obj` of class `C` is deployed. There are four different cases:
  1. `User`: the user has to enter the parameter name. This happens when only the user knows how to specify the parameter value. In this case, the automatic deployer leaves the parameter unspecified and the user will have to manually instantiate it.
  2. `Default( X )`: the parameter must be set to the default value `X`.
  3. `Req`: the parameter is required to be defined by SmartDepl: here, SmartDepl is responsible to first create an appropriate object and then pass it as parameter when `obj` is instantiated.
  4. `List(X)`: the parameter requires a list of at least `X` objects (where `X` is a natural number) that should be defined by SmartDepl. Similar to what happens with the `Req` parameter, `X` objects should be created and their list passed as parameter when `obj` is instantiated.

3 User desiderata

In this section we present how the user can specify the desired deployment. SmartDepl allows users to specifies deployment requirements as program annotations and processes them to generate, for every annotation, a new class that specifies the deployment steps to reach the desired target.

The most important part of the annotation is the definition of the constraints that the final configuration has to satisfy. In the following, before entering into the details of the SmartDepl annotation, we give an overview of the specification language to define the user desiderata.

3.1 Declarative Requirement Language

Computing a desirable deployment configuration requires taking into account the goals users expects to reach. For instance, in the Fredhopper Cloud Services, the initial goal is to deploy with reasonable cost 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 language for stating the constraints that the final configuration should satisfy.

As shown in Table 2 that reports the DRL grammar defined using the ANTLR tool,[1] a desiderata is a (possibly quantified) Boolean formula b_expr obtained 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.

[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 '[' 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 2: DRL grammar.

QueryServiceImpl[staging] > 0

The square brackets are optional (Line 12 - first option) for objects with only one default deployment scenario. Regular expression (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

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 15, 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. 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, 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)

3.2 SmartDepl Annotation

SmartDepl processes JSON like annotations. Since in ABS annotations are typed and it is not possible to enter directly a JSON annotation, the JSON annotation is written in ABS as a string of the annotation of type SmartDeploy. For example, assuming that JSON_String is the JSON string to pass to SmartDepl, the annotation can be embedded into the ABS code as follows.

[2]We assume that the user can launch only a finite 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.

```
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 3: JSON annotation example.

[SmartDeploy : JSON_String]

These annotations can be included everywhere in the ABS program. As far as the structure of the JSON object is concerned, let us introduce it with an example. Let us 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 a Query Service instance in live mode. The annotation that captures this requirement is the JSON object defined in Table 3.

In Line 1, the keyword "id" specifies that the name of the class that will be generated by SmartDepl is AddQueryDeployer. As we will see later, the user can exploit this name to scale the system, assuming the class exists. The second line contains the desired configuration in DRL, that in this case requires just the creation of a new Query Service in live mode. Deploying a new instance of the Query Service may involve other relevant objects from the surrounding environment, such as the PlatformService, or a Load Balancer in the same availability zone. Which objects are relevant 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 provided 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 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).

Currently this annotation is given in a textual form, but we are considering its generation from a more user-friendly 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`.

4 Step by Step Example

We are now ready to put all the things together and show how SmartDepl can be used to generate the deployment code and how this code can be used within the ABS program.

First of all, let us remind that 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 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 the addInstanceDescription method invoked on 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.

SmartDepl processes the ABS program and the CloudProvider specification to retrieve what kind of DC can be used and their cost. The goal of SmartDepl is then to alleviate the user with the task of deciding which machine to launch and how to deploy object on them (i.e., generate automatically the last two lines of the code above).

The Fredhopper Cloud Services use case uses the type c3_xlarge and c3_2xlarge of the Amazon EC2 instances.[3] Moreover, these may be distributed in the US or in Europe. The DC corresponding to these virtual machines can be model in ABS as follows.

```
1  CloudProvider cloudProvider = new CloudProvider("CloudProvider");
2  cloudProvider.addInstanceDescription(Pair("c3_xlarge_eu",
3    InsertAssoc(Pair(CostPerInterval,210), InsertAssoc(Pair(Memory,750),
4    InsertAssoc(Pair(Cores,4), EmptyMap)))));
5  cloudProvider.addInstanceDescription(Pair("c3_xlarge_us",
6    InsertAssoc(Pair(CostPerInterval,210), InsertAssoc(Pair(Memory,750),
7    InsertAssoc(Pair(Cores,4), EmptyMap)))));
8  cloudProvider.addInstanceDescription(Pair("c3_2xlarge_eu",
9    InsertAssoc(Pair(CostPerInterval,420), InsertAssoc(Pair(Memory,1500),
10   InsertAssoc(Pair(Cores,8), EmptyMap)))));
11 cloudProvider.addInstanceDescription(Pair("c3_2xlarge_us",
12   InsertAssoc(Pair(CostPerInterval,420), InsertAssoc(Pair(Memory,1500),
13   InsertAssoc(Pair(Cores,8), EmptyMap)))));
```

As previously stated, SmartDepl also requires to know the cost annotation for all the relevant classes. These costs can be derived by profiling the use of the different modeled entities. In the case of the Fredhopper Cloud Services every service has been profiled and its relative resource consumption has been annotated in ABS as shown in Section 2 for the QueryServiceImpl class.

After all this information are defined it is possible to write the annotation to use SmartDepl. For the Fredhopper Cloud Services use case, SmartDepl can be used twice: a first time to synthesize the initial deployment of the entire framework and a second time to dynamically add (and later remove) instances of the Query Service if the system needs to scale.

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 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 3.1. 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)
```

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

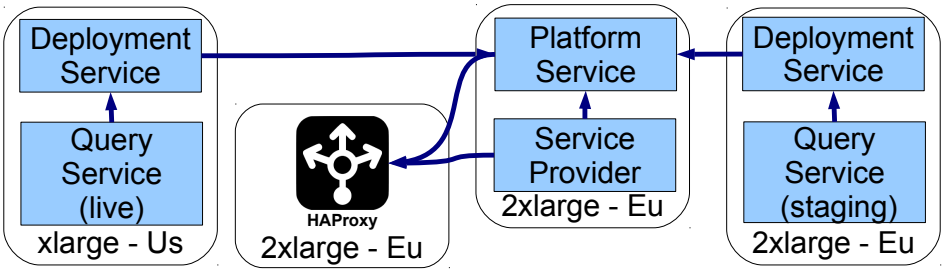

Figure 3: Example of automatic objects allocation to deployment components.

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 3.1 shows how this is expressed.

To handle catastrophic failures, the Fredhopper Cloud Services aim to balance the Query Services between the 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['.*'])

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)

Since this is the initial deployment no objects exist. Hence, the JSON SmartDepl annotation to trigger the generation of the class responsible for the deployment of the initial configuration is the following one.

```
{
  "id": "MainSmartDeployer",
  "specification": "QueryServiceImpl['staging'] > 0 and ..."
  "DC": [],
  "obj": []
}
```

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. The code to deploy this configuration is generated in the `MainSmartDeployer.deploy` method. To trigger the deployment of the initial configuration the developer has to assume the existence of such a class creating an instance of it and then execute the `deploy` method. This is done as follows.

```
1 SmartDeployInterface c1 = new MainSmartDeployer(cloudProvider);
2 c1.deploy();
```

SmartDepl generates classes implementing the interface SmartDeployInterface that is populated with the deploy, undeploy methods to perform and undo the deployment actions. Moreover, SmartDeployInterface is also populated with getter methods that can be used to retrieve the number of objects or DC that have been created by running the deploy method. For instance, after running the previous deploy method to retrieve the objects having interface IQueryService it is possible to call the method c1.getIQueryService().

After this initial deployment, the Fredhopper Cloud Services needs to scale up or down based on the traffic load. 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 it simply requires two Query services in live mode. The annotation to use in this case is exactly the one presented in Section 3.2 with the exception of the specification than now requires the deployment of two Query services in live mode and all the other constraints required by Fredhopper. 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 deploy code automatically generated by SmartDepl is the following one.

```
1  Unit deploy() {
2    DeploymentComponent c3_xlarge_eu_0 =
3      cloudProvider.prelaunchInstanceNamed("c3_xlarge_eu");
4    ls_DeploymentComponent =
5      Cons(c3_xlarge_eu_0,ls_DeploymentComponent);
6    DeploymentComponent c3_xlarge_us_0 =
7     cloudProvider.prelaunchInstanceNamed("c3_xlarge_us");
8    ls_DeploymentComponent = Cons(c3_xlarge_us_0,ls_DeploymentComponent);
9    [DC: c3_xlarge_us_0] DeploymentService
10     oDef___DeploymentServiceImpl_0_c3_xlarge_us_0 = new
11     DeploymentServiceImpl(platformObj);
12   ls_DeploymentService =
13     Cons(oDef___DeploymentServiceImpl_0_c3_xlarge_us_0, ls_DeploymentService);
14   [DC: c3_xlarge_eu_0] DeploymentService
15     oDef___DeploymentServiceImpl_0_c3_xlarge_eu_0 = new
16       DeploymentServiceImpl(platformObj);
17   ls_DeploymentService =
18     Cons(oDef___DeploymentServiceImpl_0_c3_xlarge_eu_0, ls_DeploymentService);
19   [DC: c3_xlarge_eu_0] IQueryService
20     olive___QueryServiceImpl_0_c3_xlarge_eu_0 = new
21     QueryServiceImpl(oDef___DeploymentServiceImpl_0_c3_xlarge_eu_0, "Customer
22     X", False);
23   ls_IQueryService =
24     Cons(olive___QueryServiceImpl_0_c3_xlarge_eu_0, ls_IQueryService);
25   ls_Service = Cons(olive___QueryServiceImpl_0_c3_xlarge_eu_0,
26     ls_Service);
27   ls_EndPoint = Cons(olive___QueryServiceImpl_0_c3_xlarge_eu_0,
28     ls_EndPoint);
29   [DC: c3_xlarge_us_0] IQueryService
30     olive___QueryServiceImpl_0_c3_xlarge_us_0 = new
31     QueryServiceImpl(oDef___DeploymentServiceImpl_0_c3_xlarge_us_0, "Customer
32     X", False);
33   ls_IQueryService =
34     Cons(olive___QueryServiceImpl_0_c3_xlarge_us_0, ls_IQueryService);
35   ls_Service = Cons(olive___QueryServiceImpl_0_c3_xlarge_us_0,
36     ls_Service);
37   ls_EndPoint = Cons(olive___QueryServiceImpl_0_c3_xlarge_us_0, ls_EndPoint)
38 }
```

The generated code first creates two DCs (Lines 2 and 6). After creating the DCs the internal variable ls_DeploymentComponent containing the list of all the newly created DCs is updated. Then two Deployment Services are created (Line 9 and 14). These are indeed required before the creation of the Query Services. At the end, the two Query Services are created (Lines 19 and 29). The remaining instructions update internal variables that are used to store the newly created objects.

To properly instantiate the AddQueryDeployer class we need to provide already deployed objects. As previously written, these can be retrieved by using the getter methods of the class MainSmartDeployer. In particular, based on the annotation, we need to provide a Platform service, a Load Balancer, and a Service provider. The existing services can be obtained as follows.

```
1  List<PlatformService> platformServices = c1.getPlatformService();
2  PlatformService platformService = head(platformServices);
3  List<LoadBalancerService> loadBalancerServices = c1.getLoadBalancerService();
4  LoadBalancerService loadBalancerService = head(loadBalancerServices);
5  List<ServiceProvider> serviceProviders = c1.getServiceProvider();
6  ServiceProvider serviceProvider = head(serviceProviders);
```

Now, the ABS code to scale the system up or down based on a monitor decision can be written as 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. If the monitor decides to upscale by adding new Query Services (Line 2) a new deployment decision object is created (Line 3). Here AddQueryDeployer is the name associated with the annotation previously discussed. Its first parameter is the cloud provider. 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 3. 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.[4]

SmartDepl exploits Delta Modeling to inject the generated code of the classes and methods. To compile the main program successfully the compiler should be called with the option –**product**=SmartDeploy passing as argument the main program and the delta generated by SmartDepl.

The ABS model described in this tutorial with all the annotations and specifications is available at `https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/test/FRH_staging_re.abs`.

5 Acknowledgments

To generate the code, SmartDepl relies on Zephyrus2,[5] a configuration optimizer inspired by the work conducted within the Aeolus Project.[6]

[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]`https://bitbucket.org/jacopomauro/zephyrus2`

[6]`http://www.aeolus-project.org/`