HAL
archives-ouvertes.fr

On the Integration of Automatic Deployment into the ABS Modeling Language

Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, Gianluigi Zavattaro

▶ **To cite this version:**

Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, Gianluigi Zavattaro. On the Integration of Automatic Deployment into the ABS Modeling Language. [Technical Report] Inria Sophia Antipolis. 2015. hal-01170926v2

HAL Id: hal-01170926
https://hal.inria.fr/hal-01170926v2

Submitted on 8 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Integration of Automatic Deployment into the ABS Modeling Language*

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

[1] SDL, Netherlands
[2] CWI, Netherlands
[3] Leiden University, Netherlands
[4] University of Bologna/INRIA, Italy

Abstract. In modern software systems, deployment is an integral and critical part of application development (see, e.g., the DevOps approach to software development). Nevertheless, deployment is usually overlooked at the modeling level, thus losing the possibility to perform deployment conscious decisions during the early stages of development. In this paper, we address the problem of promoting deployment as an integral part of modeling, by focusing on the Abstract Behavioral Specification (ABS) language used for the specification of models of systems composed of concurrent objects consuming resources provided by deployment components. We extend ABS with class annotations expressing the resource requirements of the objects of that class. Then we define a tool that, starting from a high-level declaration of the desired system, computes a model instance of such system that optimally distributes objects over available deployment components.

1 Introduction

Nowadays it is more and more frequent to observe an integration among the application development and deployment phases. The most popular approach in this specific context, is the one promoted by the DevOps community that aims at the automation of deployment starting from application-dependent deployment information. Modeling languages for deployment have been already proposed [8,12,18]. In this paper we take a complementary approach: we intend to investigate the integration of deployment within an existing modeling language, thus allowing for the reasoning about deployment at the application modeling level in a declarative way. Driven by a use-case considered in the ENVISAGE FP7 European Project, we integrate automatic deployment in the ABS (Abstract Behavioural Specification) language [1]. ABS has a formal semantics [14] and is

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

used to model systems based on asynchronously communicating concurrent objects distributed over deployment components that can be seen as containers offering to objects the resources they need to run.

The considered use case is given by the Fredhopper Cloud Services, which 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 costs of the virtual computing resources to acquire, and the necessity of some clients to keep their data private. These relevant aspects are considered only at deployment time and not during the application modeling and development.

We envisage several advantages from the anticipation at the modeling level of aspects related with deployment. On the one hand, this allows for an early analysis of different alternative deployments, thus providing the operation team with a valuable decisions support. On the other hand, it is possible to detect the need for additional iterations in the system design in case the results of the deployment analysis are not satisfactory. In this way, it is not necessary to test real installations of the system in order to detect design decisions having a negative impact on the system deployment.

Within the ENVISAGE project, the Fredhopper Cloud Services have been already modeled with the ABS language. This language is therefore the suitable candidate to lift for taking into account also deployment aspects. The approach that we present for integrating deployment into ABS is based on three main pillars: (i) software artifacts are enriched with the indication of their functional dependencies and the quantification of the resources they require in order to be properly executed, (ii) a high-level language for the declarative specification of the desired deployment allowing to express the minimal requirements for the desired system (e.g., the basic components that must be present or the number of replica of a given service to guarantee high availability), (iii) an automatic engine that, taking as input the local requirements of the single software artifacts and the global expectations on the desired system, computes a fully specified deployment that satisfies both kinds of constraints and minimize the total deployment costs. Summarizing, the first main contributions of the paper is the extension of ABS with the possibility to annotate class definitions with deployment information. Several deployment scenarios can be considered and, for each of them, it is possible to indicate specific functional and resource-dependent requirements. The second contribution is the definition of DDLang, a domain specific language allowing for the high-level declarative specification of the desired deployment. Moreover, we also provide an implementation of *Model-Driven Deployment Engine* (MODDE), a tool that given the set of available ABS classes (annotated with their deployment information) and the declarative specification in DDLang of the desired system, computes an ABS main program that creates the needed deploy-

ment components and deploys on them the required objects. The deployment components are taken from a description of the available computing resources (each one with an associated cost) given to MODDE as an additional input.

It is worth to mention that in the implementation of MODDE we have taken advantage of two already available tools: the configuration engine Zephyrus [4] to support the computation of the optimal allocation of objects over deployment components, and the Metis planner [16] for the generation of the sequence of actions to be executed by the generated ABS main program. We have decided to leverage on already available tools that are not tailored to a specific modeling language, to realize an easily portable and adaptable framework for model-driven deployment. In fact, if an alternative modeling language is considered instead of ABS, it will be possible to adapt our approach simply by extending that modeling language with the deployment annotations, and by modifying only those (limited) parts of MODDE that depend on ABS. Our declarative deployment language DDLang can be indeed applied to any other object-oriented modeling language as it has no particular dependencies on the specific aspects of ABS.

The paper structured as follows. In Section 2 we present the extension to the ABS modeling language for the definition of models extended with deployment information. The declarative deployment language DDLang is presented in Section 3 while Section 4 discusses the implementation of MODDE. Section 5 discuss the test of our approach on the Fredhopper Cloud Services use case. Before some concluding presented in Section 7, Section 6 discuss the related literature.

2 Annotated ABS

In this section we will briefly describe the ABS language focusing only on those aspects that are concerned with deployment: namely classes, objects instantiation, interfaces, and deployment components. Moreover, we present our extension of ABS with class annotations expressing the deployment requirements of the objects obtained as instances of such classes.

2.1 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 [1].

The basic element to capture the deployment in ABS is the *deployment component*, which is a container for objects/services.

```
DeploymentComponent small = new DeploymentComponent("m1",
    map[Pair(Memory,500), Pair(CPU,1)]);
DeploymentComponent large = new DeploymentComponent("m2",
    map[Pair(Memory,1500), Pair(CPU,4)]);
[DC: large] Service s1 = new Service();
```

```
[DC: large] Service s2 = new Service();
[DC: small] Balancer b = new Balancer(list[s1,s2]);
```

In the ABS code above, the two deployment components `small` and `large` are initially created. Every deployment component has an associated identification string and a set of provided resources. Next, three objects are created: the first two are services that are located on the `large` deployment component, while the last one is a balancer located on the `small` deployment component. Notice that the balancer receives as initialization parameters a list with the references to the two service objects. In ABS it is possible to declare interface hierarchies and define classes implementing them.

```
interface EndPoint { }
interface ReverseProxy extends EndPoint { }
class Balancer(List<Service> services) implements
    ReverseProxy { ... }
```

In the excerpt of ABS above, the `ReverseProxy` service is declared as an interface that extends `EndPoint`, and the class `Balancer` 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.

2.2 ABS annotations

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. We do not focus on pre-defined resources. In our context a resource is simply a measurable quantity that can be consumed by the ABS program. Common resources that a service can consume are memory or CPU clock cycles.

We require an annotation for every relevant class that can be involved in the automatic generation of the main program that deploys the system. 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`.

An example of an annotated ABS (i.e., the specification of the Query API service of the Fredhopper Cloud Services) is shown in Listing 1.1. 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. 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`.

```
ann
  : '[Deploy: scenario[' expr (',' expr)* ']]';
expr
  : 'Name(' STRING ')'
  | 'MaxUse(' INT ')'
  | 'Cost(' STRING ',' INT ')'
  | 'Param(' STRING ',' paramKind ')';
paramKind
  : User
  | 'Default(' STRING ')'
  | Req
  | 'List(' INT ')';
```

Table 1. Grammar of ABS annotations.

```
interface IQueryService extends Service {
  List<Item> doQuery(String q); }
[Deploy: scenario[
  MaxUse(1),
  Cost("CPU", 1), Cost("Memory", 400),
  Param("c", Default("CustomerX"),
  Param("ds", Req)]]
class QueryServiceImpl(DeploymentService ds, Customer c)
  implements IQueryService { ... }
```

Listing 1.1. Fredhopper Query API

- `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.
- `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 MODDE: here, MODDE 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 MODDE. Similar to what happens with the `Req` parameter, `X` objects should be created and their list passed as parameter when `obj` is instantiated.

Let us now consider the annotated ABS code of Listing 1.1. Abstracting away the implementation details, the Query API has been modeled as a `QueryServiceImpl` class implementing the interface `IQueryService`. The interface and the class `QueryServiceImpl` are defined in ABS at Lines 2 and 8. The annotation for the class `QueryServiceImpl` is introduced before the class definition, at Line 3. The annotation at Line 4 specifies that an object of `QueryServiceImpl` may be used as parameter only once during the creation of other objects. Line 5 associates some resource costs to an object of `QueryServiceImpl`. In particular, in this case an object of class `QueryServiceImpl` can consume up to 4GB of memory and 1 CPU. Lines 6 and 7 annotate the single initialization parameters of the class. `QueryServiceImpl` has two parameters: `ds`, an object implementing the `DeploymentService` interface, and the customer `c`. The `ds` parameter is set as a required parameter. This means that before deploying an object `obj` of `QueryServiceImpl`, it is necessary to deploy an object implementing `DeploymentService` and pass this object as initialization parameter to `obj`. The `customer` parameter is instead set to a default value, in this case `CustomerX`.

Multiple annotations are possible for the same class to identify different ways to deploy the same type of object. For instance, consider the possibility that the object of class `QueryServiceImpl` for a different customer requires 2GB of memory instead of 4GB and 2 CPUs. To capture this we can add before the class definition the following annotation.

```
[Deploy: scenario[ Name( "NewCustomer")
   MaxUse(1),
   Cost("CPU", 2), Cost("Memory", 200),
   Param("c", Default("NewCustomer"),
   Param("ds", Req) ]]
```

This annotation represents a deployment scenario identified by `NewCustomer` (Line 1) that consumes a different amount of resources and considers a different default value for the `c` parameter.[5]

3 DDLang

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 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).

[5] Please note the annotation in Listing 1.1 represents the default scenario (`Def`) since the `Name` annotation is not defined.

```
1  spec
2    : expr comparisonOP expr | spec boolOP spec | 'true' |
3    | 'not' spec | '(' spec ')' ;
4  expr
5    : 'DC[' resourceFilter '|'  simpleExpr ']'
6    | 'DC['  simpleExpr ']'
7    | expr arithmeticOP expr | simpleExpr ;
8  resourceFilter
9    : STRING comparisonOP INT
10   | resourceFilter ';' resourceFilter ;
11 simpleExpr
12   : exprNoDC comparisonOP exprNoDC
13   | simpleExpr boolOP simpleExpr |
14   | 'true' | 'not' spec | '(' spec ')' ;
15 exprNoDC :
16   INT | 'INTERFACE[' STRING ']'
17   | 'CLASS[' STRING ']' |  'CLASS[' STRING ':' STRING ']'
18   | exprNoDC arithmeticOP exprNoDC ;
19 comparisonOP : '<=' | '<' | '=' | '>=' | '>' ;
20 arithmeticOP : '+' | '-' | '*' ;
21 boolOP : 'and' | 'or' | 'impl' | 'iff' ;
```

Table 2. DDLang grammar.

All these goals and desiderata can be expressed in the *Declarative Deployment Language* (DDLang): a language for stating the constraints that the final configuration should satisfy. As shown in Table 2 that reports the DDLang grammar defined using the ANTLR tool,[6] a constraint is a specification `spec` of basic constraints `expr comparisonOP expr` (Line 2) combined using the usual logical connectives. These basic constraints specify how many elements (e.g., classes, interfaces, or deployment components) the user desires to create. An expression `expr` could identify different kinds of basic quantities: (i) an integer value, (ii) the number of objects implementing an interface `I` (denoted `INTERFACE[I]` - Line 16), (iii) the number of objects of a class `C` (denoted `CLASS[C]` - Line 17). In this last case, it is also possible to indicate the number of objects of a class `C` deployed following a given scenario `S` (`CLASS[C : S]` - Line 17).

With this expressiveness it is possible to add constraints that abstract away from the deployment components. For instance, one might require the deployment of at least 2 objects implementing the interface `IQueryService` and exactly 1 object of class `PlatformServiceImpl` by using the following expression.

```
INTERFACE[IQueryService] >= 2 and CLASS[PlatformServiceImpl]
    = 1
```

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

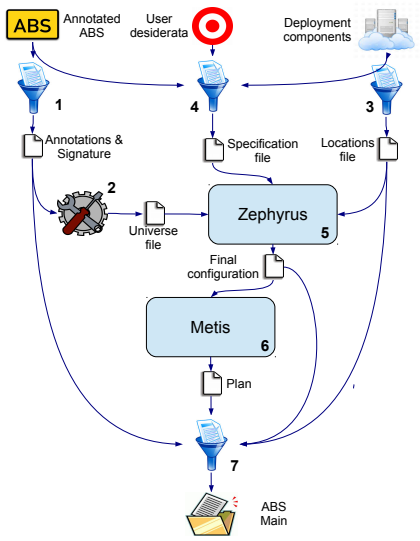

Fig. 1. MODDE execution flow

More complex quantities are concerned with deployment components. These are expressed (Line 5) with the notation `DC[ filter | simpleExpr ]` where `filter` is a sequence of constraints on the resources provided by the deployment component and `simpleExpr` is an expression. `DC[ filter | simpleExpr ]` denotes the number of deployment components that satisfy the resource constraints of `filter` and that contain objects satisfying the expression `simpleExpr`. For instance, we can specify that no deployment component having less than 2 CPUs should contain more than one object of class `QueryServiceImpl` as follows.

```
DC[ CPU <= 2 | CLASS[QueryServiceImpl] >= 2 ] = 0
```

It is interesting to notice that using such constraints it is also possible to express co-location or distribution requests. For instance, for efficiency reasons it could be convenient to co-locate highly interacting objects or, for security or fault tolerance reasons, two objects should be required to be deployed separately. For instance, in the considered case study, we require that an object of class `QueryServiceImpl` must be always co-installed together with an object of class `DeploymentServiceImpl`. This can be achieved as follows.

```
DC[CLASS[QueryServiceImpl] > 0 and CLASS[
    DeploymentServiceImpl] = 0 ] = 0
```

4 Deployment Engine

MODDE is the tool that we have implemented to generate an ABS main program realizing a deployment of objects, obtained as instantiations from a set of annotated classes, which satisfies constraints expressed in DDLang. The tool relies on

scripts that integrate Zephyrus and Metis. Zephyrus [4] is a tool that generates, starting from a description of the target application, a fully detailed architecture indicating which components are needed and how to distributed them. Metis [16] is a planner that generates a deployment plan to bring the current state of a deployed application to the new, desired one. These tools are used following the workflow depicted in Figure 1. More precisely, MODDE takes three distinct inputs: the ABS program annotated as discussed in Section 2, the user desiderata formalized as constraints in the language DDLang defined in Section 3, and the list of available deployment components expressed as described below.

The list of components is given as a JSON object having two properties: `DC_description`, which describes the different types of deployment components, and `DC_availability`, that specifies the number of available instances for each of these types. A deployment component type is identified by a name, the list of the resources it provides and a cost that the user has to pay in order to use it. For instance the following JSON object defines the possibility of using 5 `c3.large` and 3 `c3.xlarge` Amazon AWS instances as deployment components.

```
{ "DC_description": [
  { "name" : "c3.large", "cost" : 105,
    "provide_resources" : {"CPU" : 2, "Memory" : 375} },
  { "name" : "c3.xlarge", "cost" : 210
    "provide_resources" : {"CPU" : 4, "Memory" : 750} } ],
 "DC_availability": {
   "c3.large" : 5, "c3.xlarge" : 3 } }
```

The `c3.large` AWS machine is identified as a deployment component type that provides 2 CPUs and 3.75 GB of RAM. When used, this type of deployment component cost 105 credits per hour.

When MODDE is executed, the first step builds an abstract syntax tree of the annotated ABS program, retrieving all the annotations and the class signatures. This step (step 1 in Figure 1) is performed by a Java program that outputs a JSON file. In the second step, the output of the annotation extraction is processed to generate the universe file of components required by Zephyrus [4]. Zephyrus requires as input a representation of the components to deploy following the Aeolus model specification [5]. In Aeolus, a component is a grey-box showing relevant internal states and the actions that can be acted on the component to change its state during the deployment process. Each state activates provide and require ports that represent functionalities that the component offers and needs, respectively. In MODDE an ABS object `obj` is modeled as an Aeolus component with two states: an initial state `Init` representing that `obj` is not yet created, and an `On` state meaning that the object has been created. If the object has some initialization parameters requiring the existence of other objects, these are seen as require ports.[7]

The first input of the Zephyrus tool is the universe file containing all the Aeolus components obtained from the annotated classes. Moreover, to compute

[7] For more details related to the encoding of ABS objects into the Aeolus model we refer the interested reader to Appendix A.

the optimal allocation of these components, Zephyrus requires two additional inputs: a description of all locations where components can be installed and the requirements imposed on the final configuration. These two additional inputs are computed in steps 3 and 4 (see Figure 1) from the description of the deployment components and the user desiderata. In particular, in step 3, every deployment component available is translated as a Zephyrus location, associated with the resource capacities it provides. In step 4, the constraints in the DDLang input are translated into the specification request language of Zephyrus.

When all the inputs for Zephyrus are collected the solver is launched (step 5). The execution of Zephyrus is the most computation intensive task. Indeed, Zephyrus needs to solve the problem of finding the optimal allocation of the components that satisfy the user desiderata which can be seen as a generalization of the bin packing problem, a well known NP-hard problem [10]. Even though this theoretical complexity is quite high, in practice in our tested scenarios Zephyrus was able to successfully compute the optimal solution in few seconds.

Since Zephyrus can be used to minimize different quantities we use it to minimize the total cost of all the deployment components. The output of Zephyrus lists the objects that need to be deployed, where they are deployed, and their dependencies. For the generation of the ABS main program, the only remaining missing information is the deployment order of the objects creation. To get this information, in step 6, we launch Metis [16]. This planner takes in input the final configuration produced by Zephyrus and the universe file obtained at step 2 and computes the actions to be performed in order to reach the final configuration. In our specific setting where the Aeolus components have only two states, the relevant actions are the state changes from the `Init` to the `On` states.

After the generation of the Metis plan we have all the information to generate the ABS main program. The deployment components to be used are created as computed by Zephyrus. Then, following the order of the state changes computed by Metis, the new objects are created and located in the corresponding deployment components. In case an object requires other objects as initialization parameters, the required objects are passed based on the bindings among the components as defined by Zephyrus.

MODDE is written in Python (∼1k lines of code) with the exception of the annotation extractor which is written for convenience in Java (∼500 lines of code). MODDE is publicly available from https://github.com/jacopoMauro/abs_deployer.

5 Use Case

To demonstrate the feasibility of our approach, we use as a case study the deployment of the Fredhopper Cloud Services that drives over 350 global retailers with more than 16 billion in online sales every year. A typical customer 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

Metric	Value
Lines of Code	1282
Classes	13
Interfaces	16
Data Types	8
Functions	31

Table 3. Code metrics of the Fredhopper Cloud Services ABS model

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. Requests through the endpoint are then distributed over the instances. Depending on the expected number of requests from end-users or the expected service throughput, more or less instances may be deployed and be exposed through the same endpoint. This calls for specific customized deployments of the Fredhopper Cloud Services.

All the services are modeled in ABS. Table 5 summarizes the main code metrics of the Fredhopper Cloud Services ABS implementation.

To test our approach we first collected the resource consumption of instances of the most relevant classes in the ABS model. The numbers are based on real-world log files of customers of the in-production Java version of the Fredhopper Cloud Services system. CPU usage was inferred from business logs, and garbage collection logs were used to determine the memory consumption. We then associated cost annotations to the involved classes with the calculated figures. In our context, a deployment component can be considered to be an Amazon AWS instance. We defined the capacity of each resource for several AWS instance types in the locations file.[8] The price used in the cost attribute of each AWS instance type concerns on-demand instances in the US East region running Linux.[9]

We created several deployment scenarios based on the varying requirements of different customers. For instance, web shops with a large number of visitors require more Query Service instances than smaller web shops. In general, this requires a scalable, and fault tolerant system with a proportionate number of Query Service instances to handle computational tasks and network traffic and return the query results sufficiently quickly.

The deployment configuration also has to satisfy certain requirements. For instance, for security reasons, services that operate on sensitive customer data should not be deployed on machines shared by multiple customers. On the other hand, some services should be co-located with other services, for example, deploying an instance of the Query Service to a machine requires the presence of the Deployment Service on that same machine. A user can install the framework on AWS instances, exploiting the elasticity of the cloud to dynamically adapt the number of the Query Services. In the modeling of the framework,

[8] A full list of AWS instance types, with associated capacity for each resource, can be found at http://aws.amazon.com/ec2/instance-types/.

[9] http://aws.amazon.com/ec2/pricing/

the API to control the cloud resources is defined as a class that implements the `InfrastructureService` interface. Since this interface in reality is provided by Amazon itself, there is no need to deploy also an object implementing it on the customer AWS instances. To model this, we define a deployment component called `amazon_internals` that has no cost (the Amazon API is available to all its customers for free).

We have automatically generated ABS deployments for several scenarios. We report only the result obtained by MODDE when 2 instances of the Query service are required for a customer,[10] which is a simple but illustrative and common case.

```
DeploymentComponent m1.large_1 =
  new DeploymentComponent("m1.large_1", map[Pair(Memory,750),
       Pair(CPU,2)]);
DeploymentComponent m1.large_2 =
  new DeploymentComponent("m1.large_2", map[Pair(Memory,750),
       Pair(CPU,2)]);
DeploymentComponent m1.xlarge_1 =
  new DeploymentComponent("m1.xlarge_1", map[Pair(Memory,1500
      ), Pair(CPU,4)]);
DeploymentComponent m1.xlarge_2 =
  new DeploymentComponent("m1.xlarge_2", map[Pair(Memory,1500
      ), Pair(CPU,4)]);
DeploymentComponent amazon_internals =
  new DeploymentComponent("amazon_internals", map[]);

[DC: amazon_internals] InfrastructureService
  o1 = new InfrastructureServiceImpl();
[DC: m1.xlarge_1] LoadBalancerService o2 = new
    LoadBalancerServiceImpl();
[DC: m1.large_1] DeploymentService o3 = new
    DeploymentServiceImpl(o1);
[DC: m1.large_2] DeploymentService o4 = new
    DeploymentServiceImpl(o1);
[DC: m1.xlarge_2] MonitorPlatformService
  o5 = new PlatformServiceImpl(list[o3,o4], o2);
[DC: m1.large_2] IQueryService o6 = new QueryServiceImpl(o4,
    CustomerX);
[DC: m1.large_1] IQueryService o7 = new QueryServiceImpl(o3,
    CustomerX);
[DC: m1.xlarge_2] ServiceProvider o8 = new
    ServiceProviderImpl(o5, o2);
```

A graphical representation of the deployment generated by this ABS main can be seen in Figure 2. Deployment components are depicted as boxes containing

[10] The input files for MODDE implementing this use case can be found at https://github.com/jacopoMauro/abs_deployer/tree/master/test. Please note that MODDE generates long names for objects and components. Here, for the sake of brevity, we renamed these identifiers with shorter strings.

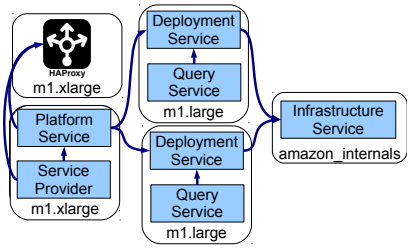

Fig. 2. Example of automatic objects allocation to deployment components.

the objects and arrows between an object a towards and object b represents the use of b as a parameter for the creation of a.

At a first sight, the deployment configuration suggested by MODDE differs from the one used in-production which uses only instances of type `c3.xlarge` (one for the Platform Service and the Service Provider, one for the Load Balancer, two for the two Query and Deployment Service pairs).

This discrepancy is due to the fact that we allowed MODDE to use all the possible AWS instances. However, Amazon is continuously updating its instances with new, better, and possibly cheaper ones. Currently, the machines of type `m1` have been deprecated and new `m1` machines could not be acquired any more. The optimal solution computed by MODDE can therefore be only used by customers that have already `m1` running machines. New customers have to rely instead on machines of type `m3` and `c3`.

If MODDE is executed taking into account just the new `m3` and `c3` AWS instances, the computed configuration obtained is exactly the one currently adopted by the operations team, thus proving its optimality.

As can be seen from this example, tool support is extremely helpful to understand what the optimal deployment scenario is in the presence of external changes, such as the appearance of new machines. With a proper estimation of the cost, using MODDE, the computation of the optimal deployment scenario is trivial and does not require a deep knowledge of the external environment conditions. This is of crucial importance because it facilitates computing the price of the final product that may vary due to external conditions such as the possibility of using (or not using) a virtual machine.

6 Related Work

The deployment of applications and services has been extensively studied in the literature. Many popular system management tools exist to that end: CFEngine [3], Puppet [15], MCollective [20], and Chef [19] are just a few among the most popular ones. Despite their differences, such tools allow to declare the components that should be installed on each machine, together with their configuration files. The burden of specifying *where* components should be deployed, and how to in-

terconnect them is left to the system administrator or cloud engineers, let alone in solving the difficult problem of optimal resource allocation.

As of today, most of the industrial products, offered by big companies, such as Amazon, HP and IBM, rely on the holistic approach where a complete model for the entire application is defined 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 [18], promoted by the OASIS consortium for open standards. TOSCA proposes an XML-like rich language (or YAML) to describe an application. Deployment plans are usually specified using the BPMN or BPEL notations, i.e., workflow languages defined in the context of business process modeling. TOSCA specifications, however, still lack proper tooling and technology support for large-scale industry cases. Following similar philosophies, but focusing more on cloud aspects, are Terraform [13], Apache Brooklyn [2], and other tools supporting the Cloud Application Management for Platforms protocol [17].

To the best of our knowledge there are no works that deal with deployment at the modeling level, providing a tool that automatically computes optimal target configurations from a declarative specification. Two recent efforts, Feinerer's work on UML [7] and Engage [9], are more similar to our approach as they both rely on a solver to plan deployments. Feinerer's work is based on the UML component model, which includes conflicts and dependencies, but lacks the aspects concerning virtual machines and deployment. Engage, on the other hand, offers no support for conflicts in the specification language. Neither Feinerer's work nor Engage allows to find a deployment that uses resources in an optimal way, minimizing the number and cost of needed (virtual) machines.

Other domain specific languages for the deployment of applications in the clouds have been proposed, e.g., the component based application model of [6], CloudML [12], and CloudMF [8]. All these approaches mainly aim at modeling the entities involved in the cloud and effective and efficient deployment engines are still to be developed for them.

7 Conclusions

In this paper we have proposed a new way to tackle and unify the modeling of a distributed system together with its deployment. We followed a model-driven approach that allows the user to specify the deployment aspects in a declarative way, without requiring in-depth knowledge of the system to be deployed. We focused and used our approach on the ABS modeling language, but we are not restricted to it: other languages such as SmartFrog [11] that have primitives to handle the deployment aspects can be used as well, provided that annotations related to the execution costs of the system are used.

We tested our approach on an industrial case study from the e-Commerce company Fredhopper. The results are encouraging since the deployment solutions resemble those (manually) devised by the operations team proving their optimality. Clearly, any automated tool that can give quicker and better eval-

uations of the deployment configuration based on a rigorous formal approach is a big step forward compared to the current practice since devising the best deployment setting is a complex, time consuming process that requires in-depth domain specific knowledge.

Based on the feedback from the operations team at Fredhopper, as future work, we will improve MODDE further addressing some of its limitations. For instance, we would like to find the best deployment configuration given a user-specified maximal cost and a maximal resource consumption. We also intend to add support for annotations with parametric costs that depend on the class parameters. Moreover, we would also like to tackle the computational aspects involved in the process of finding the optimal configuration allowing users to exploit heuristics such as local search techniques to quickly get good but possibly sub-optimal solutions.

References

1. Abstract behavioral specification language. http://www.abs-models.com/.
2. Apache Software Foundation. Apache Brooklyn. https://brooklyn.incubator.apache.org/.
3. M. Burgess. A Site Configuration Engine. *Computing Systems*, 8(2), 1995.
4. 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.
5. R. D. Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239, 2014.
6. X. Etchevers, T. Coupaye, F. Boyer, and N. D. Palma. Self-Configuration of Distributed Applications in the Cloud. In *CLOUD*, 2011.
7. I. Feinerer. Efficient large-scale configuration via integer linear programming. *AI EDAM*, 27(1):37–49, 2013.
8. 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.
9. J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, 2012.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.
11. P. Goldsack, J. Guijarro, S. Loughran, A. N. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog configuration management framework. *Operating Systems Review*, 43(1):16–25, 2009.
12. 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.
13. HashiCorp. Terraform. https://terraform.io/.
14. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO*, 2010.
15. L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1), 2006.
16. T. A. Lascu, J. Mauro, and G. Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *ICTAI*, 2013.

17. OASIS. Cloud Application Management for Platforms. http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.
18. 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.
19. Opscode. Chef. http://www.opscode.com/chef/.
20. Puppet Labs. Marionette collective. http://docs.puppetlabs.com/mcollective/.

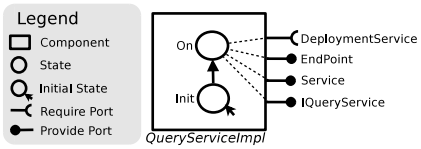

Fig. 3. Aeolus model representation for object of class `QueryServiceImpl`

A Encoding of ABS objects into Aeolus Model

An ABS object `obj` is modeled as an Aeolus component with two states: an initial state `Init` representing that `obj` is not yet created, and an `On` state meaning that the object has been created. If the object has some initialization parameters requiring the existence of other objects, these are seen as require ports. For instance, considering the code in Listing 1.1, the instantiation of an object of class `QueryServiceImpl` requires as initialization parameter an object exposing the interface `DeploymentService`. For this reason the Aeolus component representing an object `obj` of `QueryServiceImpl` requires the functionality `DeploymentService` in the `On` state. Dually, since the class `QueryServiceImpl` implements the interface `IQueryService`, the Aelous component associated to `obj` provides the functionality `IQueryService` in the `On` state, plus the interfaces `Service` and `EndPoint` which are extended by `IQueryService`. The graphical representation of this Aeolus component is reported in Figure 3. Every requirement of an interface `I` is captured as a require port in the Aeolus model, while every provided interface is captured with a provide port.

In Aeolus is possible to associate numbers to ports to deal with capacity/replication constraints. For require ports, this number indicates the minimal number of distinct components that should satisfy the requirement. Instead, for provide ports, the number stands for the maximal amount of distinct components that can use the provided functionality. In our setting, the number associated to a requirement of interface `I` for a class `C` is therefore the number of objects exposing interface `I` to be created and passed as initialization parameters to objects of class `C`. The number associated to the provide ports is instead the number defined by the `MaxUse` annotation. For example, consider an object `obj` of class `QueryServiceImpl`. The number associated to the require port `DeploymentService` is 1 since only a single object implementing the interface `DeploymentService` is needed. Moreover, since its functionality is intended to be used by only one customer (i.e., its `MaxUse` annotation is set to 1) the number associated to the provide ports is also set to 1. [11]

[11] In ABS a single class `C` can expose several interfaces (see, e.g., the three interfaces in the provide ports of Figure 3). In this case, the `MaxUse(n)` indicates the maximal usage of the object of class `C` for every single provided interface.