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

Deliverable D1.3.1
Modeling of Deployment (Initial Report)

Date of document: T18

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

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

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Modeling of Deployment (Initial Report)

This document summarises deliverable D1.3.1 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This is the first deliverable of Task T1.3 "Modeling of Deployment". This task works in strict collaboration with Task T1.2 "Modeling of Resources". In fact, deployment can be seen as the problem of allocating to computing entities the resources they need to properly run. Due to the tight connection with the activities developed within T1.2, this deliverable can be considered as a joint contribution with the deliverable D1.2.1 which contains the description of the notion of deployment component, used in ABS to characterize containers that provide objects with resources like CPU, time, bandwidth, etc.

Besides an introductory chapter, this deliverable includes three main contributions organized in three independent chapters. Chapter 2 focuses on static aspects of deployment addressing the following question: *how to model and reason about the distribution of the initial ABS objects over the deployment components that provide them with the required resources?* Chapter 3 reports about the ABS approach to the modeling of dynamic/runtime aspects of deployment: *how to model in ABS the dynamic acquisition and release of resources, as it happens nowadays in modern cloud-based applications?* A key element of the presented approach is based on the *ABS Cloud API* which provides an interface (included in the ABS Standard Library) to be used in the modeling of deployment scenarios including dynamic acquisition/release of resources. The current version of the ABS Cloud API is reported in Chapter 4.

The deliverable includes also two technical appendixes, each one containing a paper. The first one, in Appendix A and currently submitted, is integral part of Chapter 2: it describes a tool-based approach to automatic static deployment for ABS, which has been validated by applying it to the FRH case study. The second one, in Appendix B and published in the Proceedings of the 6th International ISoLA Symposium 2014, describes the application of the ABS Cloud API to the ATB case study.

List of Authors

Rudolf Schlatte (UIO)
Silvia Lizeth Tapia Tarifa (UIO)
Gianluigi Zavattaro (BOL)

Contents

Chapter 1

Introduction

In modern software systems it is more and more frequent to observe a continuum between the development and the deployment phases. For instance, the continuous delivery design practice [6] advocates the automation of the software delivery phase, in order to support the rapid and repeated releases of enhanced versions of the application. At the modeling level, such a continuum between development and deployment is far from being a common practice. In fact, traditional modeling techniques usually support the development phase (as in model-driven development approaches [12]). On the contrary, more recent modeling languages (like TOSCA [11]) focus specifically on application deployment, by expressing it in an infrastructure-independent and portable way.

To cover this gap, the Envisage project includes among its main objectives the anticipation at the modeling level of relevant aspects related to deployment. More precisely, in the D1.2.1 deliverable, the Abstract Behavioral Specification language ABS is extended to represent the basic elements characterizing deployment, namely the computing, memory, and communication resources that are usually assigned to the deployed software components. In this way, it is possible in ABS to model relevant deployment issues, as we discuss in the present deliverable, in combination with the classical ABS modeling of the architectural and behavioral aspects of an applications.

We envisage several advantages from the anticipation at the modeling level of aspects related to deployment. On the one hand, this allows for an early analysis of different alternative deployments, thus providing the operation teams, usually responsible for the actual application deployment, with a valuable decisions support. On the other hand, it is possible to detect the need for additional iterations in the system design development phase 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.

1.1 Modeling Deployment within Envisage

The Envisage project dedicates two intertwined tasks to the modeling of deployment: Task T1.2 "Modeling of Resources" and Task T1.3 "Modeling of Deployment". The deliverables of these two tasks are indeed synchronized: an initial report due at T18 and the final one at T30. For this reason, the present deliverable D1.3.1 is strictly connected with the deliverable D1.2.1. The latter is dedicated to the definition of the extensions of ABS for the modeling of resources, while the present deliverable applies such extensions to the modeling of relevant deployment scenario. For this reason, in this deliverable we assume already known (from D1.2.1) the notion of ABS deployment component and of how deployment components provide ABS objects with resources.

According to the DoW, Task 1.3 *"formalizes deployment models for virtualized architectures. . . . These models will be described in terms of combinations of several features: the amount of processing or memory resources allocated to abstract virtual machines, best and worst execution times, the choice of application-level scheduling policies for client requests, and the distribution over different abstract virtual machines with fixed* *bandwidth constraints."* When the task started (at T6), after an initial joint collaboration dedicated to the definition of commonly agreed basic modeling primitives, we organized the activities along two directions. Both are dedicated to the investigation of application-level resource allocation problems, but they focus on two distinct phases that we have called *static* and *dynamic* deployment. By static deployment we mean the initial activation/configuration of an application. In ABS terminology, static deployment is the initial creation of deployment components and localization inside those deployment components of the objects needed to complete the considered ABS model. By dynamic deployment we mean application-level scheduling policies that distribute client requests on the services that are currently available or, if needed, instantiate new services while the application is running.

We have decided to follow these two lines of investigations because, inspired by the FRH and ATB case studies, we have identified two specific independent problems that can be investigated in parallel.

The first one, taken from the FRH case study, deals with the optimization of the initial static deployment. The FRH case study in fact, consists of a system that requires customized instantiation depending on the customer profile (like the expected number of final users, the possibility to assist to usage peaks, etc.). Different system instantiations correspond to different static deployments, decided on the base of specific resource usage profiles, distinct replication criteria for specific critical sub-services, or other specific desiderata like the installation of components in the same virtual machine (or installation in distinct virtual machines) for efficiency (or fault tolerance) reasons.

Concerning the ATB case study, it includes an intensive computation –organized in smaller (independent) tasks according to the MapReduce pattern– which is usually executed in the cloud. The access to cloud computing resources is typically done via APIs allowing for a dynamic acquisition and release of virtual machines. The possibility to acquire virtual machines on-demand opens an entire spectrum of different application level policies for the management of dynamically acquired resources. On the one hand of the spectrum, the application could always use the initially acquired virtual machine. On the other hand of the spectrum, the application could acquire a new virtual machine every time a new task must be executed. Modeling in ABS these (very) different policies allows for an anticipated evaluation of their advantages/disadvantages by using the analysis or the simulation tools available for ABS.

Another motivation behind the decision to proceed in parallel in our study of static and dynamic deployment is that we envisage the possibility to combine, in a second phase, the obtained results. In fact, we expect to exploit the expertise on static resource allocation and the ability to model dynamic deployment strategies, to attack the more complex problem of dynamic acquisition, release and (re)distribution of resources based on introspection of the current state of the system. In fact, the policies for dynamic deployment that we have studied so far are agnostic of run-time information like the current level of utilization of the resources provided by a virtual machine.

1.2 Contributions and Structure of the Deliverable

Chapter 2 reports the main results obtained about the modeling and analysis of static deployment. The main contributions of this Chapter can be grouped into two categories: the identification of the relevant basic elements to be modeled, and the realization of a tool-based support for automated static deployment.

We have identified the following three elements, not present in ABS, that are relevant in the modeling of static deployment: (i) local functional and resource requirements of objects, (ii) global deployment constraints, and (iii) available types of deployment components. Concerning (i), we have extended ABS with class annotations describing properties of the objects of that class. Two kinds of properties can be expressed: the dependencies on other objects in the system to be deployed (e.g. a front-end balancer needs at least two back-end services) and the amount of required resources (like CPU units or memory). As far as (ii) is concerned, we have defined a domain specific language (DSL) to express the constraints that the expected deployment should satisfy. For instance, it is possible to require the presence of a minimum number of instances of a given type of object, or imposing the installation of objects in the same (or in distinct) deployment component(s). This first version of the DSL is not directly included into ABS because the two languages focuses on distinct aspects: ABS is expected to contain the specification of local deployment information associated to the declared ABS classes, while the DSL is used to express global requirements on the desired system deployment. The possible inclusion of this language into ABS will be subsequently evaluated and in case reported in the final deliverable of Task T1.3. Finally, concerning (iii), we have defined a specific format for describing the different kinds of available deployment components, the corresponding provided resources, and their cost.

Once all these elements have been specified, we have implemented a tool that synthesizes an ABS main procedure so as to realize its optimal deployment, i.e. instantiates a group of deployment components and of interconnected objects that satisfy the expressed requirements and constraints, having the minimum possible cost. This represents a new model-driven approach to declarative automated deployment whose details are reported in the technical annex in Appendix A.

Chapter 3 reports about the Envisage approach to the modeling of dynamic deployment policies. The idea is to model the dynamic acquisition/release of computing resources by means of ABS classes implementing a specific interface offering methods that, for instance, return new deployment components, allow for the release of such components, or return the current accumulated costs. Such an interface is referred to as the ABS Cloud API. A simple example of modeling of a cloud API in ABS is discussed in Chapter 3 by means of a couple of examples. The first one deals with a generic cloud-based system with clients issuing requests either at constant rate or according to an irregular distribution including peaks. The services run in the cloud, and several deployment strategies are modeled. Among these strategies, a constant strategy is considered that initially acquires the computing resources that will be used to run the services for the entire system life-time. Another considered strategy takes decisions at run time: new computing resources are dynamically acquired if all those that are already available are currently busy. The second one deals with a more realistic system: the Montage toolkit for generating science-grade mosaics by composing multiple astronomical images [7]. Montage is a modular system subject to different deployment scenarios spanning from installation on researcher's desktop machine to more advanced deployments on a grid or a cloud.

Once the different deployment strategies have been modeled, we can use the ABS simulation tool to compare their performances (remaining at the modeling level). This approach has been already applied to an initial modeling of the *MapReduce* part of the ATB case study (see the technical annex in Appendix B).

Chapter 4 reports the description of the version of the ABS Cloud API currently present in the ABS Standard Library. The API aims to integrate and systematize the common parts of the case studies, especially regarding lifecycle management of the ABS deployment components that are used to model virtual and physical machines. As such, the API forms a common base on which the case studies can experiment with deployment strategies, resource management, cost vs. time optimization, etc. We will assess (and in case we will amend and extend) the API by means of its application to the modeling of the case studies. For this reason, the current ABS Cloud API should be considered a preliminary version; the final version will be reported in D1.3.2 (due at T30). Another related point that will be discussed in D1.3.2 is the identification of a failure model, obtained by refining what has been initially presented at T12 in D1.1, to address dynamic deployment failures like, for instance, the interruption of an Amazon EC2 Spot Instance. More advanced aspects of dynamic deployment, indeed, will be addressed in the next year of activity and reported in Deliverable D1.3.2.

Chapter 2

Automatic Static Deployment

In this chapter we report the Envisage approach to the modeling and analysis, by means of the ABS language, of the initial "static" application deployment. By static deployment we mean the initialization of an ABS model, consisting of the creation of the needed deployment components, and the activation inside such deployment components of the initial objects. These initial deployments are usually specified by the ABS *main* procedures, that contain the commands for constructing the initial deployment components and the objects initially located inside them. In extreme synthesis, in this chapter we present a tool-based approach for the automatic generation of main procedures that realize static deployments that, on the one hand, are guaranteed to provide to each initial object sufficient resources and, on the other hand, are optimal according to costs associated to deployment components.

Our main source of inspiration for the present work has been provided us by the FRH case study, in particular the problem of customizing the deployment of instances of the Fredhopper Cloud Services based on the customer profile (e.g. the expected number of final clients, possible usage peaks, etc.). Reasoning about deployment at the modeling level can have several interesting benefits. For example, in the case of Fredhopper Cloud Services, it can be a valuable support to the decisions currently taken by the so-called *operations team* responsible to actually deploy the Fredhopper Cloud Services instances.

The approach we have followed to deal with static system deployment is based on three main pillars that, to be as general as possible, here we present independently of their application to the ABS modeling language. The first one deals with the modeling of the software artifacts composing the desired system: their description is enriched with the indication of their functional dependencies and the quantification of their required resources. The second one consists of a high-level language for the declarative specification of the desired deployment: minimal requirements can be expressed on the system to be deployed like, for instance, the basic components that must be present (e.g., the need for a load balancer) or the number of replica of a given service to guarantee for instance high availability. The third pillar is 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 specific deployment that satisfies both kinds of constraints and possibly optimizes some objective function to minimize the total deployment costs.

Beyond driving our research, the FRH case study has been used to validate the results of our work.

Contributions of the chapter The main contributions of this chapter are as follows.

- 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 for the objects obtained as instantiation of such classes.
- The definition of DDLang, a domain specific language allowing for the high-level declarative specification of the desired deployment.
- The 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 deployment 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 in JSON format.

It is worth to mention that in the implementation of our tool MODDE we have taken advantage of two already available tools, namely Zephyrus [4] and Metis [10], to respectively support the computation of the optimal allocation of objects over deployment components, and the generation of the sequence of actions to be executed by the generated ABS main program. We have decided to leverage 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 such a 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.

Structure of the chapter In Section 2.1 we present the extension to ABS for the description of deployment information. The declarative deployment language DDLang is presented in Section 2.2 while Section 2.3 discusses the implementation of MODDE. Sections 2.4 discusses the validation of our approach to model-driven deployment.

2.1 Annotated ABS

Ideally, we would like to have a measure of the resource consumption associated to every object that can be created, in such a way that we can have an 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. For instance, common resources that a program can consume are memory, CPU clock cycles, and bandwidth. The resource amount is expressed with a natural number. For instance, assuming that the minimal unit to measure the RAM memory is a MB, we can state that a deployment component provides 2GB of RAM simply by associating to a given deployment component $1024 * 2$ units of memory. We associate to objects their expected maximum amount of needed resources, and when two objects requires an amount r_1 and r_2 of the same resource we assume that the cumulative consumption does not exceed the sum of r_1 and r_2. Obviously, a resource can never be consumed in more quantities than provided.

We require an annotation for every relevant class that can be involved in the automatic generation of the main. For instance, there is no need to annotate a class implementing an internal data structure. 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` (for instance, to avoid to overload a database instance, one could impose a maximum number of services that in the deployed system use that instance).

The grammar of the annotation language can be found in the technical annex (Appendix A), here we simply report a couple of examples. The first one is in Listing 2.1, taken from the specification of the Query API of Fredhopper Cloud Services.

Abstracting away the implementation details, the Query API has been modelled as a `QueryServiceImpl` class implementing the interface `IQueryService`. The interface and the class `QueryServiceImpl` are defined in ABS at Lines 1 and 9. The annotation for the class `QueryServiceImpl` is introduced before the class definition, at Line 4. The annotation `MaxUse(1)` at Line 5 specifies that an object of `QueryServiceImpl` may be used by only one client object; technically speaking, the main can pass only once the reference to this object to other objects in the deployment. Line 6 associates some resource costs to an object

```
1  interface IQueryService extends Service {
2     List<Item> doQuery(String q);
3  }
4  [Deploy: scenario[
5     MaxUse(1),
6     Cost("CPU", 1), Cost("Memory", 4096),
7     Param("c", Default("CustomerX"),
8     Param("ds", Req)]]
9  class QueryServiceImpl(DeploymentService ds,
10   Customer c) implements IQueryService {
11    // Implementation
12 }
```

Listing 2.1: FRH Query API

of `QueryServiceImpl`. In particular, in this case an object of class `QueryServiceImpl` can consume up to 4GB of memory and 1 CPU. Lines 7 and 8 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 the string `CustomerX`.

As mentioned above, 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.

```
1  [Deploy: scenario[ Name( "NewCustomer")
2     MaxUse(1),
3     Cost("CPU", 2), Cost("Memory", 2048),
4     Param("c", Default("NewCustomer"),
5     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.[1]

2.2 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).

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. The syntax of DDLang is reported in the technical annex (Appendix A), here we discuss the ideas behind the language and some simple examples. DDLang is used to express constraints on two kinds of quantities: basic quantities dealing with object cardinalities, and more complex quantities dealing with deployment components cardinalities. Concerning basic quantities, it is possible to express the number of objects exposing a given interface, or deployed according to a predefined scenario. For instance, it is possible to require the presence of at least 3 instances of `IQueryService` among which at most 1 deployed according to the `NewCustomer` scenario.

[1]Please note the annotation in Listing 2.1 represents the default scenario (that we implicitly identify with `Def`) since the scenario name is not explicitly indicated.

Complex quantities are used to express the number of deployment components such that their resources satisfy some given constraints and the contained objects satisfy some other (basic) constraints. For instance, it is possible to express that at least one deployment component should provide 2 units of `CPU`, and no deployment component can contain more than two instances of `QueryServiceImpl`.

More precisely, constraints on basic quantities are of kind `expr comparisonOP expr`, that can be combined using the usual logical connectives. The syntactic element `comparisonOP` is a classical comparison operator. The 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]`), (iii) the number of objects of a class `C` (denoted `CLASS[C]`). 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]`).

With this expressivity 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
```

More complex quantities are concerned with deployment components. These are expressed with the notation `DC[ filter | simpleExpr ]` where `filter` is an optional 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] > 1 ] = 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
```

The impossibility to co-locate two objects in the same deployment component can be expressed in a similar manner. For example, in the FRH case study, we require that `PlatformServiceImpl` and `LoadBalancerServiceImpl` are installed separately for fault tolerance reasons. This requirement is captured by the following constraint.

```
DC[ CLASS[PlatformServiceImpl] > 0 and
    CLASS[LoadBalancerServiceImpl] > 0 ] = 0
```

2.3 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 following the workflow depicted in Figure 2.1. More precisely, MODDE takes three distinct inputs: the ABS program annotated as discussed in Section 2.1, the user desiderata formalized as constraints in the language DDLang presented in Section 2.2, 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

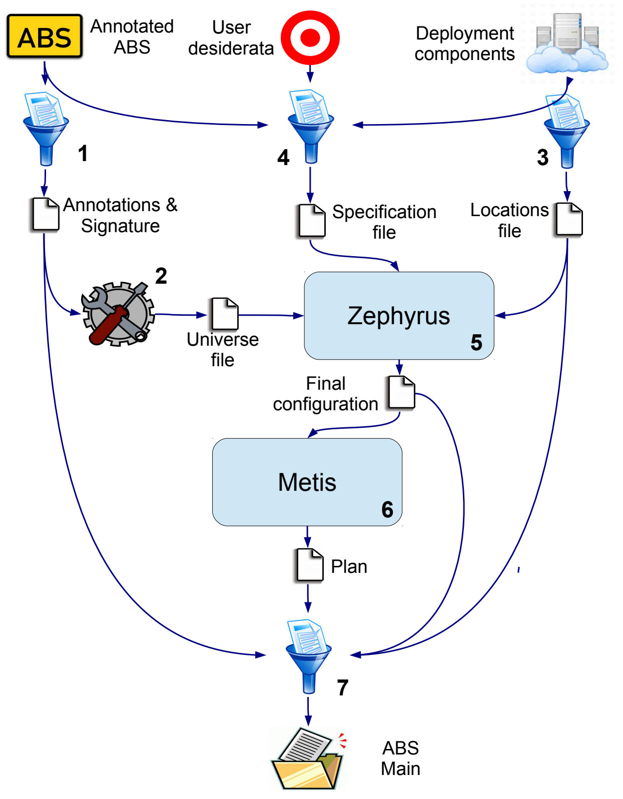

Figure 2.1: MODDE architecture

instances for each of these types. A deployment component type is identified by a name, the list of the resources it provides and a (monetary) 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.

```json
{
 "DC_description": [
  { "name" : "c3.large",
    "provide_resources" :
      {"CPU" : 2, "Memory" : 3750},
    "cost" : 105 },
  { "name" : "c3.xlarge",
    "provide_resources" :
      {"CPU" : 4, "Memory" : 7500},
    "cost" : 210 } ],
 "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.

The internals of the MODDE are detailed in the technical annex (Appendix A). Here, we simply comment the tool architecture reported in Figure 2.1. When MODDE is executed, the first step builds a parse-tree of the annotated ABS program, retrieving all the annotations and the class signatures. This step (step 1 in Figure 2.1) 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 the tool Zephyrus. In fact, Zephyrus requires the description of the software artifacts to combine as components equipped with provide and require ports. Besides the component universe, 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 2.1) from the description of the deployment components and the user desiderata.

When all the inputs for Zephyrus are collected the solver is launched (step 5). Since Zephyrus can be used to minimize different quantities we use it to minimize the total cost of all the deployment components used. The output of Zephyrus lists the objects that need to be deployed, the locations where they must be deployed, and their bindings between require and provide ports.

For the generation of the ABS main program, the only remaining missing information is the deployment order of the objects that need to be installed. To get this information, in step 6, we launch Metis. This planner takes in input the final configuration produced by Zephyrus and, given the ABS annotations describing the functional inter-object dependencies, it computes the precise order in which objects should be created in order to reach the final configuration. 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 deployment actions 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. More precisely, if an object has require ports, it receives as initialization parameters the references to those objects exposing the provide ports to which it has been connected by Zephyrus.

2.4 Validation

In order to validate 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 AWS instance. We defined the capacity of each resource for several AWS instance types in the locations file.[2] The price used in the cost attribute of each AWS instance type concerns on-demand instances in the US East region running Linux.[3]

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 (and this varies over time: visitor peaks are typically observed around Christmas or during promotions). 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. Below we list some of these requirements of the Fredhopper Cloud Services.

- Platform Service and Service Provider should be located in the same location, but no other Services should reside at the same location, and there is only one instance of Platform Service (shared by all customers).
- Load Balancer should not be co-located with other Services and is dedicated per customer (for large customers, there may be multiple Load Balancers).

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

[3] `http://aws.amazon.com/ec2/pricing/`

- Query Service should always be deployed together with the Deployment Service on a dedicated machine (per customer).

Section 2.2 shows the formal versions of some of the above requirements. The specification language proved to be sufficiently expressive to capture the above and all other requirements.

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 modelling of the framework, 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) and is used to deploy the object implementing the Amazon interface.

We have automatically generated ABS deployments for several scenarios. We report and comment only the result obtained by MODDE when 2 instances of the Query service are required for a customer, which is a simple but already significative case.[4]

```
DeploymentComponent m1.large_1 =
  new DeploymentComponent("m1.large_1",
    map[Pair(Memory,7500), Pair(CPU,2)]);
DeploymentComponent m1.large_2 =
  new DeploymentComponent("m1.large_2",
    map[Pair(Memory,7500), Pair(CPU,2)]);
DeploymentComponent m1.xlarge_1 =
  new DeploymentComponent("m1.xlarge_1",
    map[Pair(Memory,15000), Pair(CPU,4)]);
DeploymentComponent m1.xlarge_2 =
  new DeploymentComponent("m1.xlarge_2",
    map[Pair(Memory,15000), 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);
```

[4] MODDE generates long names for objects and components. Here, for the sake of brevity, we renamed these identifiers with shorter strings.

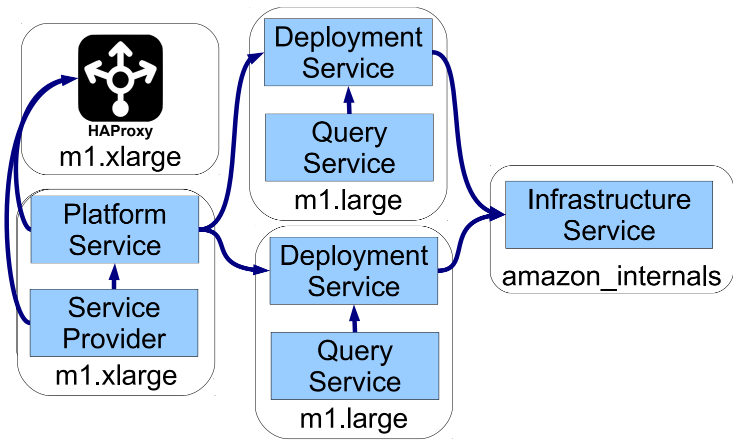

Figure 2.2: Example of automatic objects allocation to deployment components.

A graphical representation of the deployment generated by this ABS main can be seen in Figure 2.2. Deployment components are depicted as boxes containing 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 costumers that have already `m1` running machines. New costumers 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 extremely important 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.

Chapter 3

Patterns for Dynamic Deployment Strategies in ABS

A common strategy for web applications these days, especially in early development and deployment, is to acquire the needed resources (server, storage, bandwidth) from a cloud infrastructure provider such as Amazon, Windows or Google, instead of purchasing server hardware and data centre space. In that way, initial costs can be kept low while still keeping the flexibility to react quickly to demand growth [2].

This chapter collects examples of how resource management can be integrated in ABS models of resource-aware applications. In these examples we are integrating the resource management strategies in the client layer (see Figure 3.1 which is taken from the DoW). These examples use a simple and initial version of a Cloud API which only focus on computer resources and were originally developed in [8, 9]. The examples have been adapted to the current syntax of the toolchain implemented in the Envisage project. Section 3.1 introduces the cloud API that the examples in this chapter use. Section 3.2 describes an example of a general web application which distributes user requests to servers deployed on the cloud. Section 3.3 describes a model of the Montage toolkit deployed on the cloud. Note that Chapter 4 introduces and explains the current version of the cloud API that the case studies of the Envisage project use, which is a different API from the one presented in this chapter.

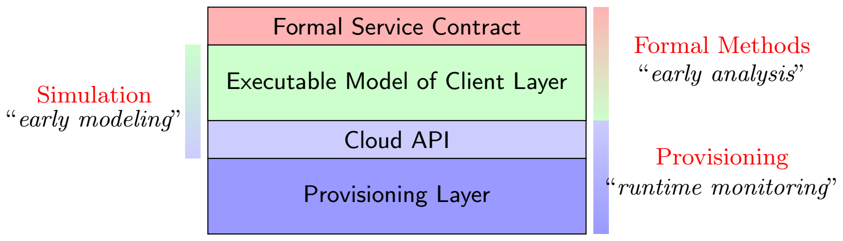

Figure 3.1: The approach to modeling services in Envisage.

3.1 Example: An Initial Model of a Cloud Provider API

In this section we describe a cloud provider infrastructure that rents *virtual servers* to its clients. This particular cloud API focuses on CPU resources, for this reason, the cost of leasing a virtual server depends on the configuration of the CPU resources. In this initial and simple cloud API, we assume that it is in the interest of the client application that virtual servers are kept running only when they are busy processing requests from users, and that they are stopped and returned to the cloud provider otherwise. Furthermore, we assume here that creating a machine is instantaneous. Note that these assumptions have changed in later versions of the cloud API.

Figure 3.2 shows the interface of the cloud provider. The `CloudProvider` API includes methods for

```
interface CloudProvider {
    DeploymentComponent createMachine(Int capacity);
    Unit acquireMachine(DeploymentComponent machine);
    Unit releaseMachine(DeploymentComponent machine);
    Rat getAccumulatedCost();
}
```

Figure 3.2: Interfaces of the cloud provider.

creating, acquiring and releasing virtual machines. In the implementation, this is done by creating deployment components on which the client application can deploy objects. Using these operations, an application *clients* can create Server objects on virtual machines (modeled by deployment components) that are obtained from the CloudProvider via the method createMachine. Once the virtual machines are created and Server objects are deployed, the applications can use the methods acquireMachine and releaseMachine to start and stop virtual machines, so that the objects created inside them can process requests.

In addition, the cloud provider keeps track of the accumulated costs incurred by the client application. For this simple API, the cost is calculated in terms of the sum of the processing capacities of the *active* virtual machines; i.e., a call to acquireMachine(dc) starts accounting for the virtual machine dc and a call to releaseMachine(dc) stops the accounting again for dc. The method getAccumulatedCost returns the accumulated cost of the client application. Inside the cloud provider, an active run method does the accounting for every time interval. For more details about the implementation if this cloud provider, see [9].

3.2 Example: Application-Level Management of Virtualized Resources

In this example we model and analyse by means of simulations a general web application which distributes user requests to a number of servers deployed on the cloud. To clarify terminology, in this section we shall refer to the clients of the web service as *users*, and the clients of the cloud provider (such as the web service) as *clients*. Our aim here is to model and analyse a cloud-enabled application. As part of the model of the application, we include a component which handles the management of virtualized resources at the application level. This component monitors the user demand, provisions servers as needed, and distributes user requests between the active servers in order to meet the deadlines of the user requests while keeping the costs of leasing virtual servers down.

The example depicted in Figure 3.3 is a model of a *client* application which interacts with a *cloud provider API* from Section 3.1 and with a *user*. This client application consists of a (dynamic) number of *servers* and one *balancer* which is the main focus of our case study. The balancer is in charge of the management of the virtualized resources acquired by the client application. The user sends processing requests to the balancer, which sends them to an active server. To keep the focus on the balancer, we do not model the details of these requests; instead, they carry a deadline and a processing cost that represent an abstraction of QoS and computing requirements.

It is the responsibility of the balancer to implement a resource management strategy which both minimizes the cost of running the client application on the cloud and maximizes the application's QoS (i.e., minimizes the number of deadline misses for user requests). Note that this model does not aim for precise measurements, but rather for a rough understanding of the system behavior by means of simulations. Hence, no precise costs of running the system are obtained via the simulations (which would depend on the varying price of CPU hours). Rather, *different balancing strategies* can be compared by evaluation against different usage scenarios, for example a user with a *steady request rate* or with an unexpected increase of *load spike*.

Figure 3.4 shows the interfaces of the entities of the case study in Real-Time ABS (the user needs no interface since it is not referenced by any object). Each Server has a method process, which incurs run-time costs on the server's deployment component, which can be found via the getDC method. The Balancer's request method is called from the User. The balancer is responsible for creating Server objects on the CloudProvider, using the API from Section 3.1, so that the Server objects can process requests.

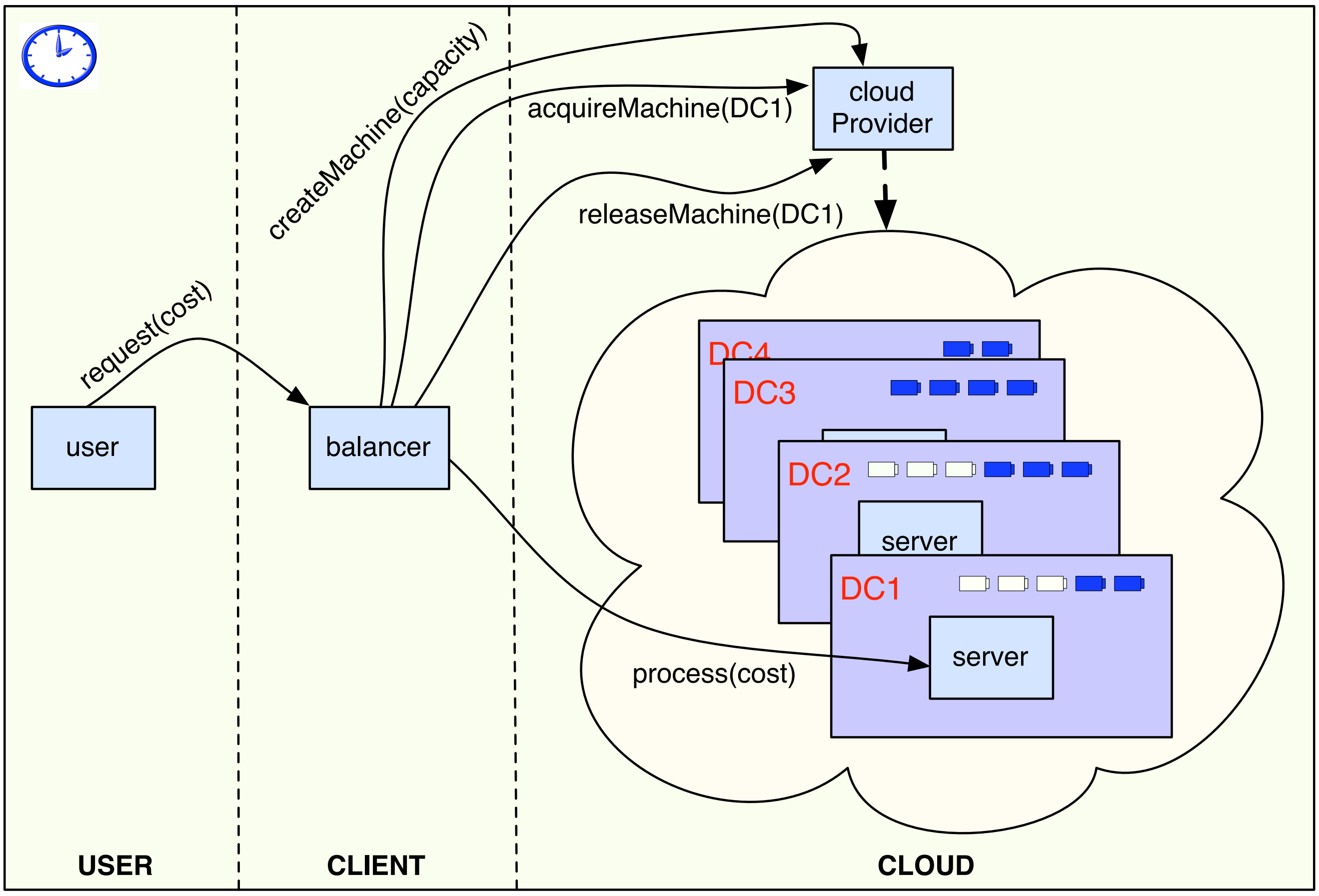

Figure 3.3: An on-demand deployment architecture for the client application. Neither user nor cloud provider contribute to the cost of running the system, and we assume that the request processing costs are significantly bigger than the CPU resources needed to run the balancer. Hence, only the servers are running in dedicated virtual machines.

```
interface Server {
    Unit process(Int cost); // called from Balancer
    DeploymentComponent getDC();
}

interface Balancer {
    Bool request (Int cost); // called from Client
}
```

Figure 3.4: Interfaces of the case study.

The Server . The class `Server` shown in Figure 3.5, which do not change as we vary strategies and user behavior, implements the Server interface and is quite straightforward. The method `process` consumes resources according to its cost argument, and the method `getDC` simply returns the deployment component on which the server object is deployed.

The User Scenarios. We consider two user scenarios: *steady load* and *load spike*. The two scenarios are modeled by the corresponding classes SteadyLoadUser and LoadSpikeUser, given in Figure 3.6. The two classes have fields numRequests and numFailures, which are used for counting the number of sent requests

```
class Server implements Server {
    Unit process (Int cost) {
        while (cost > 0) { [Cost: 1] skip; cost = cost − 1}
    }

    DeploymentComponent getDC() { return thisDC();}
}
```

Figure 3.5: Implementation of the Server class.

and the number of missed deadlines for these requests, respectively. Both classes implement the method sendRequest which calls request with a given deadline on the balancer, suspends execution while waiting for the reply to the call, and does the bookkeeping after the reply has been received by incrementing the fields numRequests and numFailures as appropriate. The frequency of these requests is controlled by the active run method which differs between the two classes. In the SteadyLoadUser class, the run method asynchronously calls sendRequest and then suspends for a fixed duration. In contrast the run method of LoadSpikeUser has the same steady load behavior except for a window of time (between time 60 and 80 according to the clock), during which there is a load spike in which asynchronous calls to sendRequest are sent with much shorter intervals.

3.2.1 Balancing strategies

In this case study, we model three different balancers for the application-level management of the virtualized resources. The balancers provide the front end to our web application, which receives user requests, and uses backend servers, deployed on the cloud, for processing these user requests. The different balancers reflect different strategies for interacting with the cloud provider to achieve the resource management, and may be described as follows:

- the **constant balancer** simply allocates one server sufficient for the expected load and keeps it running;
- the **as-needed balancer** calculates the server size needed to fulfill a specific request within the deadline, and allocates the needed resources disregarding the cost; and
- the **budget-aware balancer** operates with a given budget of CPU resources per time unit. Unused CPU resources can be "saved for later" to cope with unexpected load spikes, but the cost of running the system is still bounded.

The Constant Balancer. captures over-provisioning by processing all requests on a single server which should have sufficient capacity, and is modeled by the class ConstantBalancer in Figure 3.7. It initializes the web application by requesting a single machine from the cloud provider, on which it deploys a concurrent object group consisting of a Server object. After initialization, the constant balancer uses this server to process all user requests, and returns success to a user request if it was processed within the deadline.

The As-Needed Balancer. is modeled by the class DynamicBalancer in Figure 3.8. This class maintains a data structure sleepingMachines which sorts available machines (with deployed servers) by CPU processing capacity. We omit the (straightforward) definitions of the following auxiliary functions on this data structure: hasMachine(s,i) checks if a machine of capacity i is available in the structure s; addMachine(s,i,m) adds a machine m to the set associated with capacity i in s; and removeMachine(s,i,m) removes the machine m from the set associated with i in s.

When the DynamicBalancer receives a request, it calculates the machine capacity resources needed to fulfill the request, and requests a server deployed on a machine of appropriate size by calling the method

```
class SteadyLoadUser(Balancer b) {
    Int numRequests = 0;
    Int numFailures = 0;
    Unit run() {
        while (True) {
            this!sendRequest();
            await duration(5, 5);
        }
    }

    Unit sendRequest() {
        [Deadline: Duration(2)] Fut<Bool> s = b!request(3);
        await s?;
        Bool success = s.get;
        numRequests = numRequests + 1;
        if (¬success) numFailures = numFailures + 1;
    }
}

class LoadSpikeUser(Balancer b) {
  Int numRequests = 0;
  Int numFailures = 0;
  Unit run() {
    while (True) {
      if (timeValue(now()) > 60 && timeValue(now()) < 80) {
        this!sendRequest();
        await duration(1, 1);
      } else {
        this!sendRequest();
        await duration(5, 5);
      }
    }
  }

  Unit sendRequest() {
    [Deadline: Duration(2)] Fut<Bool> s = b!request(3);
    await s?;
    Bool success = s.get;
    numRequests = numRequests + 1;
    if (¬success) numFailures = numFailures + 1;
  }
}
```

Figure 3.6: Different user behavior modeled by the two classes SteadyLoadUser and LoadSpikeUser.

this.getMachine(resources). When it gets the server, it asynchronously calls process on this server and suspends. Once the reply is available, it calls **this**.dropMachine(server) and returns success to the user if the processing happened within the deadline.

The method getMachine first checks in sleepingMachines if there are available servers deployed on machines of appropriate size, in which case such a server is returned. (The auxiliary function take(s) selects an element of the set s.) Otherwise, the balancer requests a new machine from the cloud provider by calling createMachine and deploys a server on the new machine. The method dropMachine asks the cloud provider to stop running the machine on which the server is deployed and returns the server to the sleepingMachines set of appropriate capacity. The field costPerTimeUnit keeps track of the amount of resources *currently leased from the cloud provider*, and is updated by both methods getMachine and releaseMachine. This is the amount of resources for which the application is currently charged.

The Budget-Aware Balancer. is a resource management strategy in which the balancer has a certain *budget per time interval*, and may save resources for later. This balancer is modeled by the class

```
class ConstantBalancer(CloudProvider provider, Int serverSize) implements Balancer {
    Server server;
    DeploymentComponent dc;
    Bool initialized = False;

    Unit run() {
        dc = await provider!createMachine(serverSize);
        [DC: dc] server = new Server();
        initialized = True;
    }

    Bool request (Int cost) {
        await initialized;
        Fut<Unit> r = server!process(cost); await r?;
        return (durationValue(deadline()) > 0);
    }
}
```

Figure 3.7: The Real-Time ABS model of the constant balancer.

```
class DynamicBalancer (CloudProvider provider) implements Balancer {
    Map<Int, Set<Server>> sleepingMachines = EmptyMap;
    Int costPerTimeUnit = 0;
    Int machineStartTime = 0;

    Server getMachine(Int size) {
        Server server = null;
        Time t = now();
        costPerTimeUnit = costPerTimeUnit + size;
        if (hasMachine(sleepingMachines, size)) {
            server = take(lookupUnsafe(sleepingMachines, size));
            sleepingMachines = removeMachine(sleepingMachines, size, server);
            DeploymentComponent dc = await server!getDC();
            Fut<Unit> fa = provider!acquireMachine(dc); await fa?;
        } else {
            DeploymentComponent dc = await provider!createMachine(size);
            [DC: dc] server = new Server();
        }
        machineStartTime = timeDifference(t, now());
        return server;
    }

    Unit dropMachine(Server server) {
        DeploymentComponent dc = await server!getDC();
        Fut<Unit> fr = provider!releaseMachine(dc); await fr?;
        InfRat size = await dc!total(CPU);
        costPerTimeUnit = costPerTimeUnit − finvalue(size);
        sleepingMachines = addMachine(sleepingMachines, finvalue(size), server);
    }

    Bool request (Int cost) {
        Int resources = (cost / durationValue(deadline())) + 1 + machineStartTime;
        Server server = this.getMachine(resources);
        Fut<Unit> r = server!process(cost); await r?;
        this.dropMachine(server);
        return durationValue(deadline()) > 0;
    }
}
```

Figure 3.8: The Real-Time ABS model of the as-needed balancer.

BudgetBalancer in Figure 3.9, with a class parameter budgetPerTimeUnit which determines this budget, and a field availableBudget which keeps track of the accumulated (saved) resources. The fields sleepingMachines, costPerTimeUnit, and machineStartTime and the methods getMachine and dropMachine are as in the previous example of the DynamicBalancer class. When the budget-aware balancer gets a request, it calculates the resources needed to fulfill the request in the variable wantedResources and the resources it has available on the budget in maxResources. If there are resources available on the budget, the budget-aware balancer calls getMachine to get the best server the request according to the budget. The budget-aware balancer has an active run method which monitors the resource usage and updates the available budget for every time interval. It also maintains a log budgetHistory of the available resources over time.

```
class BudgetBalancer(CloudProvider provider,Int budgetPerTimeUnit) implements Balancer {
  Map<Int, Set<Server>> sleepingMachines = EmptyMap;
  Int costPerTimeUnit = 0;
  Int machineStartTime = 0;
  Int availableBudget = 1;
  List<Int> budgetHistory = Nil;

  Unit run() {
        while (True) {
      availableBudget = availableBudget + budgetPerTimeUnit − costPerTimeUnit;
      budgetHistory = Cons(availableBudget, budgetHistory);
      await duration(1, 1);
    }
  }

  Bool request(Int cost) {
    Bool result = False;
    Int wantedResources = (cost / durationValue(deadline())) + 1 + machineStartTime;
    Int maxResources = (budgetPerTimeUnit − costPerTimeUnit)
          + (max(availableBudget, 0) / durationValue(deadline()));
    if (maxResources > 0) {
      Server server= this.getMachine(min(wantedResources,maxResources));
      Fut<Unit> r = server!process(cost); await r?;
      this.dropMachine(server);
      result = (durationValue(deadline()) > 0);
    }
    return result;
  }

  Server getMachine(Int size) { ... } // as in the DynamicBalancer
  Unit dropMachine(Server server) { ... }// as in the DynamicBalancer
}
```

Figure 3.9: The Real-Time ABS model of the budget-aware balancer.

3.2.2 Comparing Balancing Strategies

Real-Time Maude has a formally defined semantics [1] which is used to implement a model simulator in the Maude system [3]. In order to compare the three balancing strategies of our case study, we simulate their behavior for the two user scenarios described in Section 3.2, in each case with a single "user" object generating requests. For simplicity, we here set the budget of the budget-aware balancer to 1. All simulations were run for 100 units of simulated time. The following measurements were extracted from the simulation traces:

- *quality of service* measured as the number of successful requests (i.e., requests completed within the deadline) divided by the total number of requests; and
- *accumulated cost* of running the machines, measured as the total sum of CPU resources made available by the cloud provider.

Table 3.1 summarizes the results. Not surprisingly, the as-needed balancer exhibits the best QoS numbers, but at potentially unbounded runtime cost. The constant balancer with a single running server exhibited both the highest runtime cost and the worst QoS under unexpected load with the chosen scenarios.

	User scenario			
	Steady load		*Load spike*	
Strategy	QoS	Cost	QoS	Cost
Constant balancer	100%	200	53%	200
As-needed balancer	100%	80	100%	128
Budget-aware balancer	100%	80	68%	97

Table 3.1: Simulation results.

The budget-aware strategy exhibits only slightly better QoS characteristics under load than the constant balancer approach, which reflects how the budget was chosen. Figure 3.10 shows the available and used budget over time. It can be seen that the available budget is mostly used during normal load, so there are not many saved resources which can be used to deal with the load spike between time 60 and 80. A more realistic system would have a monitoring component to alert an operator, who would be able to manually add budget or switch to other balancing strategies, but this functionality was not considered in our case study.

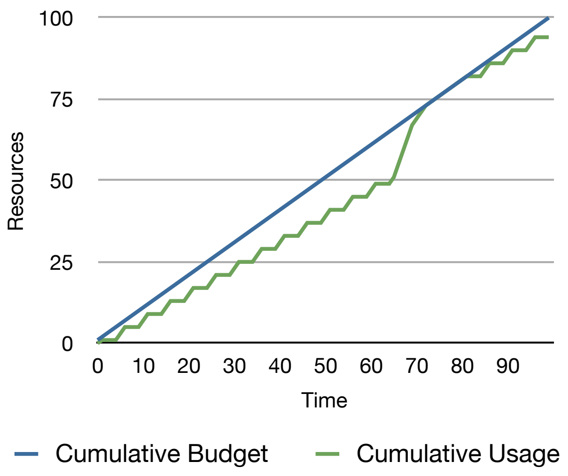

Figure 3.10: Budget use over time for the budget-aware balancer. The load spike between time 60 and 80 quickly consumes the saved-up funds.

3.3 Example: The Montage Toolkit

Montage is a portable software toolkit for generating science-grade mosaics by composing multiple astronomical images [7]. Montage is modular and can be run on a researcher's desktop machine, in a grid, or on a cloud. Due to the high volume of data in a typical astronomical dataset and the high resolution of the resulting mosaic, as well as the highly parallelizable nature of the needed computations, Montage is a good candidate for cloud deployment.

This section describes the architecture of the Montage system and how it can be modeled in Real-Time ABS. We explain how costs are associated to the different parts of the model.

Module	Description
mImgtbl	Extract geometry information from a set of FITS headers and create a metadata table from it.
mOverlaps	Analyze an image metadata table to determine which images overlap on the sky.
mProject	Reproject a FITS image.
mProjExec	Reproject a set of images, running *mProject* for each image.
mDiff	Perform a simple image difference between a pair of overlapping images.
mDiffExec	Run *mDiff* on all the overlap pairs identified by *mOverlaps*.
mFitplane	Fit a plane (excluding outlier pixels) to an image. Used on the difference images generated by *mDiff*.
mFitExec	Run *mFitplane* on all overlapping pairs. Creates a table of image-to-image difference parameters.
mBgModel	Modeling/fitting program which uses the image-to-image difference parameter table to interactively determine a set of corrections to apply to each image to achieve a "best" global fit.
mBackground	Remove a background from a single image
mBgExec	Run *mBackground* on all the images in the metadata table.
mAdd	Co-add the reprojected images to produce an output mosaic.

Figure 3.11: The modules of the Montage case study.

3.3.1 The Problem Description

Creating a mosaic from a set of input images involves a number of tasks: first reprojecting the images to a common projection, coordinating system and scale, then rectifying the background radiation in all images to a common flux scale and background level, and finally co-adding the reprojected background-rectified images into a final mosaic. The tasks exchange data in the format FITS, which encapsulates image data and meta-data. These tasks are implemented by a number of Montage modules [7], which are listed and described in Fig. 3.11. These modules can be run individually or combined in a workflow, locally or remotely on a grid or a cloud. Fig. 3.12 depicts the dataflow dependencies between the modules in a typical Montage workflow [5]. These dependencies show which jobs can be parallelized on multiprocessor systems, grids, or cloud services.

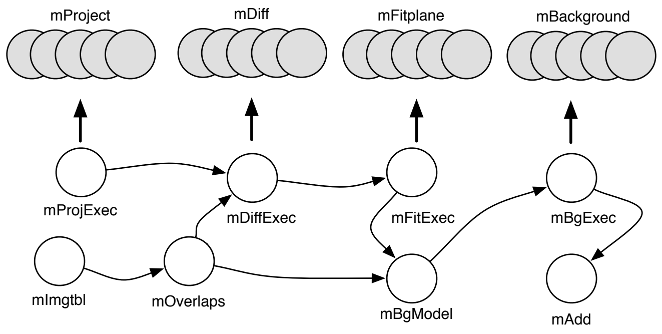

Figure 3.12: Montage abstract workflow.

We model an abstract workflow architecture of Montage toolkit. In particular, we consider the case in which Montage processes multiple input images in parallel. Our model abstracts from the implementation details of the manipulation of images, replacing them with abstract statements and cost annotations.

3.3.2 A Model of the Montage Workflow in Real-Time ABS

```
interface CalcServer {
  DeploymentComponent getDC();
  MetadataT mImgtbl(List<FITS> i);
  MetadataT mOverlaps(MetadataT mt);
  FITS mProject(FITS image);
  FITSdf mDiff (FITS image1, FITS image2);
  FITSfit mFitplane (FITSdf df);
  CorrectionT mBgModel(Image2ImageT diffs, MetadataT ovlaps);
  FITS mBackground (Int correction,FITS image );
  FITS mAdd (List<FITS> images); }

class CalcServer implements CalcServer {
  ...
  FITS mBackground (Int correction,FITS image ){
    [Cost: 1] FITS result = correctFITS(image,correction);
    return result;
  }
... }
```

Figure 3.13: CalcServer interface and class in Real-Time ABS.

The Core Modules. The core modules that execute atomic tasks (i.e., mProject, mDiff, mFitplane, mBgModel, mBackground, mAdd, mImgtbl, and mOverlaps) are modeled as methods inside a class CalcServer which implements the CalcServer interface shown in Fig. 3.13. In the methods of this class, cost annotations are used to specify the costs of executing atomic tasks. The images considered in the case study have a constant size, so it is sufficient to use a constant cost for the atomic tasks. Lacking precise cost estimates for the individual tasks, we consider an abstract cost model in which each atomic task is assigned the cost of 1 resource. (This cost model could be further refined; although some timing measurements are given in [7], these are not detailed enough for this purpose.) The code for one such atomic task inside the CalcServer class is shown in Fig. 3.13.

Resource Management. The workflow process does not interact with the different instances of CalcServer directly. Instead, tasks are sent to an instance of ApplicationServer which acts as a broker for the preallocated machine instances created in the init method. The ApplicationServer interface, partly shown in Fig. 3.14, provides the workflow which starts the parallelizable tasks (i.e., mProjExec, mDiffExec, mFitExec and mBgExec) and distributes the atomic tasks (e.g., mDiff) to instances of CalcServer. Atomic tasks are sent directly to one calculation server. Two fields activeMachines and servers keep track of the number of active jobs on each created machine and the order in which servers get jobs, respectively. Surrounding every call to a calculation server the auxiliary methods getServer and dropServer do the bookkeeping and resource management of the virtual machines. Methods getServer and dropServer call the methods acquireMachineOfObject and releaseMachineOfObject respectively, which communicate with the cloud provider API (described in Section 3.1) to acquire and release virtual machines. Asynchronous method calls to the future variables fimage and fnewimages, and task suspension are used to keep the application server responsive.

Our model defines algebraic data types FITS, FITSdf, FITSfit, as well as the list MetadataT and the maps CorrectionT and Image2ImageT to represent the input and output data at the different stages of the workflow; for example, FITS is a data type which represents image archives in FITS format, which is constructed from an abstract representation of metadata and of image data. This data can be used to keep track of data flow and abstractions of calculation results. The empty list and map are denoted Nil and EmptyMap. On lists, the constructor Cons(h, t) takes as arguments an element h and a list t; head(Cons(h, t)) $= h$ and tail(Cons(h, t)) $= t$. The function isEmpty(l) returns true if l is the empty list. On maps, the function lookupDefault(m, k, v) returns the value bound to k in m if the key k is bound in m, and otherwise it returns the default value v.

```
interface ApplicationServer {
  FITS mAdd (List<FITS> images);
  List<FITS> mProjExec(List<FITS> images);
  List<FITSdf> mDiffExec (MetadataT metatable, List<FITS> images);
  Image2ImageT mFitExec(List<FITSdf> dfs);
  List<FITS> mBgExec (CorrectionT corrections, List<FITS> images);
  ... }

class ApplicationServer(CloudProvider provider) implements ApplicationServer {
  List<CalcServer> servers = Nil; Map<DC,Int> activeMachines = EmptyMap;

  Unit init() {
     Int i = ...// number of machines deployed in the cloud;
     while (i > 0) {
        Fut<DeploymentComponent> fdc = provider!createMachine(3);
        DeploymentComponent dc = fdc.get;
        [DC: dc]CalcServer p = new CalcServer();
        servers = Cons(p, servers); i = i − 1; } }
  ...
  List<FITS> mBgExec(CorrectionT corrections,List<FITS> images) {
    List<FITS> newimages = Nil;
    if (isEmpty(images)==False) {
      FITS image = head(images);
      Int correction = lookupDefault(corrections,getId(image), 0);
      CalcServer b = this.getServer();
      Fut<FITS> fimage = b!mBackground (correction,image);
      Fut<List<FITS>> fnewimages=this!mBgExec(corrections,tail(images));
      await fimage?; FITS tmpimage = fimage.get;
      this.dropServer(b);
      await fnewimages?; List<FITS> newtmpimages = fnewimages.get;
      newimages = Cons(tmpimage, newtmpimages);}
    return newimages;}
    ...

    //Bookkeeping and resource management of the VM
   CalcServer getServer() {
      nTasks = nTasks + 1;
      await (¬(servers == Nil));
      CalcServer s = head(servers); servers = tail(servers);
      this.acquireMachineOfObject(s);
      servers = appendright(servers, s); //round−robin
      return s; }

    Unit dropServer(CalcServer s) {
      this.releaseMachineOfObject(s); lastTask = now(); }

    Unit acquireMachineOfObject(CalcServer o) {
      Fut<DeploymentComponent> fdc = o!getDC();
      await fdc?; DeploymentComponent dc = fdc.get;
      if (¬contains(keys(activeMachines), dc)) {
        Fut<Unit> f = provider!acquireMachine(dc); await f?;}
      activeMachines = incrementCount(activeMachines, dc); }

    Unit releaseMachineOfObject(CalcServer o) {
      Fut<DeploymentComponent> fdc = o!getDC();
      await fdc?; DeploymentComponent dc = fdc.get;
      activeMachines = decrementCount(activeMachines, dc);
      if (nTasks == 200) {
        Set<DC> machines = keys(activeMachines);
        while (~(emptySet(machines))) {
          DC machine = take(machines);
          machines = remove(machines, machine);
          provider!releaseMachine(machine); } } }
... }
```

Figure 3.14: The ApplicationServer interface and class (abridged).

Chapter 4

The ABS Cloud API

This chapter describes the facilities for modeling cloud deployment as currently implemented in ABS. The scope and content of the Cloud API is expected to change as the project progresses; the final version will be reported in D1.3.2.

4.1 Standard Library Support

This section builds upon Deliverable D1.2.1, where we discussed resource modeling and its effects on model simulation. This section shows support for modeling complex deployment scenarios in ABS and how this is applied in the case studies.

All ABS identifiers (classes, interfaces, functions, data types) mentioned in this chapter are exported from the module ABS.DC.

4.1.1 Datatypes, Expressions and Deployment Component Configurations

As mentioned in Deliverable D1.2.1, deployment components are involved in modeling resource configurations and deployment scenarios. All cogs and their processes are deployed on some deployment component, which will restrict execution capacity according to its resource configuration. This section expands on the use of deployment components.

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

Resource Configurations. The datatype ResourceType, as described in Deliverable D1.2.1, has constructors for the resource types in use in ABS. Currently, the resource types are CPU, Bandwidth and Memory. A *resource configuration* assigns numeric values to a subset of these resource types. Resource configurations can be used to describe, create and query deployment configurations.

Example:

```
def Map<Resourcetype, Rat> amazonSmallInstance() =
  map[Pair(CPU, 20), Pair(Memory, 10000)];
```

This example defines an amazonSmallInstance to be a deployment component with 20 CPU and 10000 Memory capacity. Note that there is no value given for bandwidth; in this case, bandwidth is deemed to be either infinite or not necessary for purposes of the given model.

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

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

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

4.1.2 The DeploymentComponent Interface

As described in Deliverable D1.2.1, cogs are deployed on deployment components. A deployment component is created with a given resource configuration which influences the non-functional properties of all cogs deployed thereon.

Example:

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

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

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

Example:

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

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

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

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

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

Example:

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

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

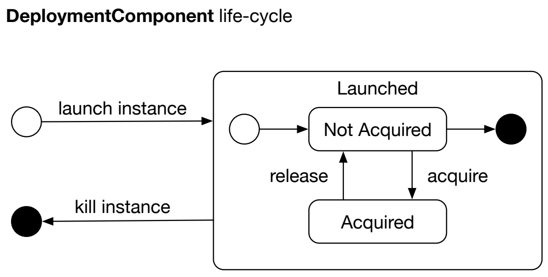

Figure 4.1: The deployment component lifecycle

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

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

4.1.3 The CloudProvider Interface and Deployment Component Lifecycle

When going beyond static scenarios with a fixed number of deployment components, it is necessary to introduce components to manage the creation, allocation, deallocation and destruction of deployment components. Such a component can also collect the billing (cost) information that results in quantitative measurements on fitness of different deployment scenarios. (See [5] for an example of such a scenario).

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

Finding a cloud provider. The DeploymentComponent interface includes a method getProvider() that returns a reference to the cloud provider that handles the given deployment component. Note that the returned reference can be **null** if the deployment component is not managed by a cloud provider.

Acquiring a deployment component. Acquiring a deployment component follows Figure 4.1. It is, however, necessary to perform launching and acquiring of instances atomically, otherwise trying to acquire a freshly-launched instance might fail in case the cloud provider handed out the fresh instance in response to another instance request that matched its resource configuration.

```
DeploymentComponent launchInstance(Map<Resourcetype, Rat> description);
Bool acquireInstance(DeploymentComponent instance);
Bool prelaunchInstance(DeploymentComponent instance);
```

The launchInstance method returns either a fresh deployment component or a deployment component that has been released previously and whose resource configuration fits the description. This method might not return instantly, modeling the fact that machine instances might not be created instantly, depending on instance type and cloud provider platform. The returned deployment component is considered to be acquired by the requestor and can be used immediately for deploying new cogs.

The acquireInstance method acquires a deployment component, i.e., after this method returns True the caller is allowed to deploy cogs on the deployment component until the deployment component is released again. If this method returns False, the deployment component has already been acquired.

Since acquiring a deployment component requires having a reference already, the DeploymentComponent interface offers a convenience method Bool acquire() with the same semantics as acquireInstance. In case the deployment component is not managed by a cloud provider, acquire will always succeed.

The method prelaunchInstance always creates a new deployment component whose resource configuration fits the description. This method can be used for load balancing purposes, or in anticipation of incoming launchInstance requests, which will experience less delay when pre-launched deployment components are already created.

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

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

After releaseInstance, a subsequent call to launchInstance might return a reference to that same deployment component if it fits. A deployment component that has been released representsx a running but idle virtual machine instance.

The DeploymentComponent interface offers a convenience method Bool release(). In case the deployment component is not managed by a cloud provider, release will always succeed.

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

4.2 Multiple Cloud Providers

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

The cloud provider of the current deployment component can be obtained via **thisDC**()getProvider()!. There is no built-in way of obtaining a reference to another cloud provider; references to multiple cloud providers are passed along in the normal way via method calls or class initialization parameters.

4.3 Application in the Case Studies

All case studies use ABS cogs and deployment components to model code deployed on virtual machines. This section briefly discusses the state of cloud deployment modeling in the case studies as of M18.

The Fredhopper case study (Task T4.3) contains an early version of the presented Cloud API (the InfrastructureService interface and InfrastructureServiceImpl class). Appendix B presents a publication describing the cloud processing part of the Atbrox case study (Task T4.2) in the form of a dynamic deployment model implementing the MapReduce processing framework. The ETICS case study of Engineering (Task T4.4) implements a version of the Cloud API in the class ResourcePool.

Work is currently under way to unify the three case studies and unify deployment component lifecycles using the common framework.

Bibliography

[1] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[2] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

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

[5] Ewa Deelman, Gurmeet Singh, Miron Livny, G. Bruce Berriman, and John Good. The cost of doing science on the cloud: The Montage example. In *Proceedings of the Conference on High Performance Computing (SC'08)*, pages 1–12. IEEE/ACM, 2008.

[6] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[7] Joseph C. Jacob, Daniel S. Katz, G. Bruce Berriman, John Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas A. Prince, and Roy Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, July 2009.

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

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

[10] Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *FACS*, 2013.

[11] 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`.

[12] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

Glossary

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

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

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

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

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

Domain Specific Language Programming or modeling language specialized for a particular application domain.

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

MapReduce Programming model for processing large data sets. Based on the idea of two kinds of parallelizable tasks it is specifically tailored for execution in a multi-threaded system.

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

Appendix A

Model-driven Declarative Deployment

Model-driven Declarative Deployment[*]

Stijn de Gouw
SDL/CWI
sgouw@sdl.com

Michael Lienhardt
University of Bologna/INRIA
michael.lienhardt@inria.fr

Jacopo Mauro
University of Bologna/INRIA
jmauro@cs.unibo.it

Behrooz Nobakht
Leiden University/SDL
bnobakht@liacs.nl

Gianluigi Zavattaro
University of Bologna/INRIA
gianluigi.zavattaro@unibo.it

ABSTRACT

Production of modern software systems frequently adopts a so-called continuous delivery approach, according to which there is a continuum between the development and the deployment phases. Nevertheless, at the modelling level, the combination between development and deployment is far from being a common practice. In this paper, we address the problem of promoting deployment as an integral part of modelling. To this aim, we adopt the object-oriented ABS language, which supports the modelling of systems composed of concurrent objects running inside deployment components. We extend ABS with class annotations expressing the requirements to be satisfied in order to deploy an object of that class. Then, we define a declarative deployment language and implement a tool that, starting from a high-level declaration of the desired system, computes a main program that instantiates an optimal deployment of the system.

1. INTRODUCTION

In modern software systems it is more and more frequent to observe a continuum between the development and the deployment phases. For instance, the continuous delivery design practice [18] advocates the automation of the software delivery phase, in order to support the rapid and repeated releases of enhanced versions of the application. At the modelling level, such a continuum between development and deployment is far from being a common practice. In fact, traditional modelling techniques usually support the development phase (see, for instance, model-driven development approaches [29]). On the contrary, more recent modelling languages (like, for instance, TOSCA [25]) focus specifically on application deployment, by expressing it in an infrastructure-independent and portable way.

In this paper, we address the problem of promoting deployment as an integral part of modelling. The approach that we present is based on three main pillars. The first one deals with the modelling of the software artefacts composing the desired system: their description is enriched with the indication of their functional dependencies and the quantification of the resources they require in order to be properly executed. The second one consists of a high-level language for the declarative specification of the desired deployment: minimal requirements can be expressed on the system to be deployed like, for instance, the basic components that must be present (e.g., the need for a load balancer) or the number of replica of a given service to guarantee for instance high availability. The third pillar is an automatic engine that, taking as input the local requirements of the single software artefacts and the global expectations on the desired system, computes a fully specified deployment that satisfies both kinds of constraints and possibly optimizes some objective function to minimize the total deployment costs.

Our research has been driven and validated by an industrial use case: the Fredhopper Cloud Services which offers search and targeting facilities on a large product database to e-Commerce companies. Depending on the specific profile of an e-Commerce company —like the expected number of clients or the preference between a completely externalized cloud-based installation or a hybrid on-premises/cloud configuration— 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 customers to keep their data private.

We envisage several advantages from the anticipation at the modelling 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.

We apply our approach to a specific modelling language, the Abstract Behavioral Specification language (ABS). ABS supports the modelling of distributed systems represented as a network of deployment components, which are containers providing concurrent asynchronously communicating ob-

[*]Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

jects with the resources they need to properly execute. The selection of ABS is justified by two main reasons: the presence in ABS of the linguistic elements needed to properly model aspects related with deployment, and the already available modelling in ABS of the Fredhopper Cloud Services. This permits the comparison of the model of deployment that our approach will automatically compute with those that are actually adopted by the operation team. In other terms, Fredhopper already has a set of concrete benchmarks that we can use to validate and evaluate the results of our model-based automatic deployments.

The main contributions of the paper are as follows.

- The extension of ABS with the possibility to annotate class definitions with deployment information. Several deployment scenarii can be considered and, for each of them, it is possible to indicate specific functional and resource-dependent requirements for the objects obtained as instantiation of such classes.

- The definition of DDLang, a domain specific language allowing for the high-level declarative specification of the desired deployment.

- The 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 deployment 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 in JSON format.

It is worth to mention that in the implementation of our tool MODDE we have taken advantage of two already available tools, namely Zephyrus [6] and Metis [22], to respectively support the computation of the optimal allocation of objects over deployment components, and the generation of the sequence of actions to be executed by the generated ABS main program. We have decided to leverage already available tools that are not tailored to a specific modelling language, to realize an easily portable and adaptable framework for model-driven deployment. In fact, if an alternative modelling language is considered instead of ABS, it will be possible to adapt our approach simply by extending such a modelling 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 modelling language as it has no particular dependencies on the specific aspects of ABS.

Structure of the paper.

In Section 2 we discuss the related literature. The description of the Fredhopper Cloud Services used to drive and validate our work is reported in Section 3. In Section 4 we present the extension to the ABS modelling language allowing for the definition of ABS models extended with deployment information. The declarative deployment language DDLang is presented in Section 5 while Section 6 discusses the implementation of MODDE. Sections 7 discuss the validation of our approach to model-driven deployment. Finally, in Section 8 we draw some concluding remarks.

2. RELATED WORK

The deployment of applications and services has been extensively studied in the literature. Automated approaches have been developed already, but thus far mostly for the particular case of configuring *package-based* FOSS (Free and Open Source Software) distributions on a *single* system. There are nowadays generic, solver-based component managers for this task [1]. Similar approaches have been developed in the context of Software Product Lines where a correct instance of a product needs to be composed of a consistent set of features [28].

Things get more complicated when the deployment of applications needs to be performed on a pool of distributed and interconnected machines. This problem has lately attracted significant attention in the area of system administration. Many popular system management tools exist to that end: CFEngine [5], Puppet [20], MCollective [27], and Chef [26] 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 interconnect them is left to the system admin, let alone in solving the difficult problem of optimal resource allocation.

Two deployment approaches standing at opposite sides are gaining more and more momentum: the *holistic* and the *DevOps* one. In the former, one defines a complete model for the entire application and the deployment plan is then derived in a top-down manner. In the latter, put forward by the DevOps community, an application is deployed by assembling available components that serve as the basic building blocks. This emerging approach works in a bottom-up direction: from individual component descriptions and recipes for installing them, an application is built as a composition of these recipes.

As of today, most of the industrial products, offered by big companies, such as Amazon, HP and IBM, rely on the holistic approach. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [25], 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 modelling. TOSCA specification, however, still lacks proper tooling and technology support for large-scale industry usages.

The most important representative for the DevOps approach is instead Juju [19], by Canonical. It is based on the concept of *charm*: the atomic unit containing a description of a component. This description in the form of meta-data is coupled with configuration data and *hooks* that are scripts to deploy and connect components. Unfortunately, even in this case, in order to use Juju, some advanced knowledge of the application to install is mandatory. This is due to the fact that the meta-data are written to support the system administrators in their decisions but are not sufficiently detailed to support completely automatic deployment. Following this philosophy, but focusing more on cloud aspects, are

Terraform [16], Apache Brooklyn [4], and other tools supporting the Cloud Application Management for Platforms protocol [24].

Recently, to overcome the limitations of the holistic and DevOps approaches, Zephyrus [6] has been introduced. This tool automatically generates, starting from a partial and abstract description of the target application written in the Aeolus Model language [7], a fully detailed architecture indicating which components are needed to realize such an application, how to distributed them on virtual machines, and how to bind them together [6]. Zephyrus is also capable of producing optimal architectures, minimizing the amount of needed virtual machines while still guaranteeing that each software component has its needed share of computing resources (CPU power, memory, bandwidth, etc.) on the machine where it gets deployed. As shown in [8], Zephyrus could be used to compute a plan of deployment steps leading to an optimal and safe configuration if used in combination with Metis [21,22]: a planner that generates a complete deployment plan that will have to be executed to bring the current state of a deployed application to a given final configuration. Plans are made of individual deployment actions like installing a software component, changing its state according to its component life-cycle, provisioning virtual machines, etc.

Inspired by the results presented in [8] where Zephyrus and Metis are used to actually deploy complex systems on an OpenStack cloud, in this work we apply them at the modelling level to the ABS object-oriented language. In this way, on the one hand, we rely on already established tools for quickly generate the desired deployment solution and, on the other hand, we develop a framework based on independent engines that can be adapted to other modelling languages.

To the best of our knowledge there are no other works that deal with deployment at the modelling level, providing a tool that automatically computes optimal target configurations. Two recent efforts, Feinerer's work on UML [10] and Engage [12], 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 of needed (virtual) machines.

ConfSolve [17] improves on the automatic component allocation: it relies on a constraint solver to propose an optimal allocation of virtual machines to servers, and of applications to virtual machines. It relies on an object-oriented declarative language (which, differently from ABS does not deal with behavioural aspects), but it does not devise a plan of actions leading to the deployment of the target and optimal configuration.

Other domain specific languages for the deployment of applications in the clouds have been proposed, e.g., the component based application model of [9], CloudML [15], and CloudMF [11]. All these approaches mainly aim at modelling the entities involved in the cloud and effective and efficient deployment engines are still to be developed for them.

As far as the modelling languages are concerned, in this work we just focus on ABS. Our findings however can be

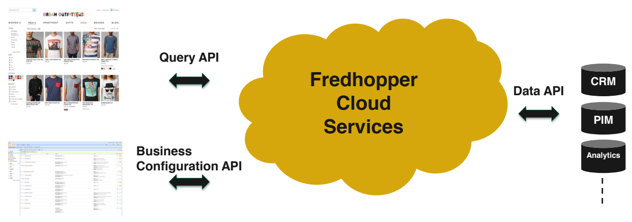

Figure 1: The Fredhopper System

reported in a rather straightforward manner to other languages. For instance, an interesting candidate language is SmartFrog [14] (a Java based language and framework developed at HP for managing deployment in a distributed setting) or its extensions such as DADL (Distributed Application Description Language) [23]. Note that all these languages cannot be used right away since, as we had to do with ABS, they need to be enriched with annotations describing the resource consumption of the various entities defined.

3. FREDHOPPER CLOUD SERVICES

Fredhopper develops 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). In addition to the cloud offering, the Fredhopper system can instead also be deployed on-premise at the customer.

The Fredhopper Cloud Services 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. Figure 1 shows a high-level view of the Fredhopper system from the customer perspective. An example of a very commonly used Fredhopper service is the Fredhopper Query API, which allows users to query over their product catalogue via full text search[1] and faceted navigation.[2]

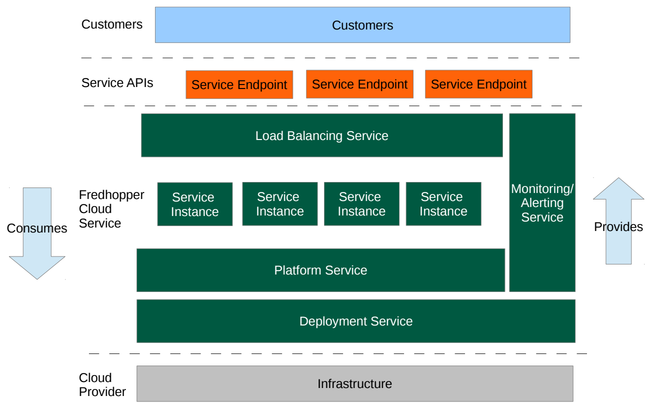

Figure 2: Architecture of the Fredhopper Cloud Services

The architecture of the Fredhopper Cloud Services is shown

[1] http://en.wikipedia.org/wiki/Full_text_search
[2] http://en.wikipedia.org/wiki/Faceted_navigation

in Figure 2. The services offered by Fredhopper are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Typically, software services are deployed as *service instances*. Each instance offers the same service and is exposed via the Load Balancing Service, which in turn offers a service endpoint. 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.

The key services of the Fredhopper Cloud Services are the following ones.

Load Balancing Service.

The Load Balancing Service is responsible for distributing requests from service endpoints to their corresponding instances. Currently at Fredhopper, this service is implemented by HAProxy:[3] a TCP/HTTP load balancer that also provides HTTP authentication.

Platform Service.

The Platform Service provides an *interface* to the Cloud Engineers to deploy and manage service instances and to expose them through service endpoints. The Platform Service takes a service specification, which includes a *resource configuration* for the service, and creates and deploys the specified service. A service specification from a customer determines which type of service is being offered, the number of service instances to be deployed initially and the amount of *virtualised resources* to be consumed by instance.

Deployment Service.

The Deployment Service provides an API to the Platform Service to deploy service instances onto specified virtualised resources provided by the *Infrastructure Service*. The API also offers operations to control the life-cycle of the deployed service instances. The Deployment Service allows the Fredhopper Cloud Services to be independent of the specific infrastructure that underlies the service instances.

Infrastructure Service.

The Infrastructure Service offers an API to the Deployment Service to acquire and release virtualised resources. At the time of writing the Fredhopper Cloud Services utilizes virtualised resources from the Amazon Web Services, [4] where processing and memory resources are exposed through Elastic Compute Cloud instances.[5]

Query Service.

The Query Service provides the basic functionality for customers to query over their product catalogue via full text search and faceted navigation. The result of a query is a list of items that satisfy the given query, ordered by decreasing relevance.

All these services are modelled in ABS. Table 3 summarizes the main code metrics of the Fredhopper Cloud Services ABS implementation.

[3]www.haproxy.org
[4]aws.amazon.com
[5]https://aws.amazon.com/ec2/instance-types/

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

Table 1: Code metrics of the Fredhopper Cloud Services ABS model

Please note that to deliver high-quality digital shopping experiences to end-users, it is crucial to find an optimal deployment configuration: the number and kind of virtual machines used in a deployment must be sufficiently powerful and the cost of the virtual machines must be maintained at an acceptable level. The deployment configuration must also take into account several requirements: some services can be shared between various customers, while other services that manipulate private customer data should be deployed on a dedicated (per-customer) basis.

Finding an optimal deployment configuration that satisfies all requirements is a complex task. It is currently done manually by an operations team. This requires domain-specific knowledge and is prone to human-error. Furthermore, the operations team takes conservative precautions to ensure customer quality, by overspending on the deployment configuration. In this context, a tool based on a rigorous formal approach that helps evaluating and finding better deployment configurations, at a fraction of the time currently required by the operations team, clearly represents a significant breakthrough.

4. 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.

4.1 ABS

The Abstract Behavioral Specification language ABS has been designed to develop executable models with an object-oriented program flow. ABS targets distributed and concurrent systems by means of concurrent object groups and asynchronous method calls. Moreover, ABS supports a range of techniques for model exploration and analysis, based on formal semantics. The reader interested in the details of ABS and the related tools can refer to the ABS project website;[6] here we simply discuss specific linguistic features supporting deployment modelling. The basic element is the deployment component, which is a container for objects.

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

[6]http://abs-models.org

```
[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 code above, `ReverseProxy` 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. As commented above, the initialization parameters of class `Balancer` consist of the list of the service instances to be balanced.

4.2 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, assuming that computing the composition of such costs is possible, 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 consume by the ABS program. For instance, common resources that a program can consume are memory, CPU clock cycles, and bandwidth. The resource amount is expressed with a natural number. For instance, assuming that the minimal unit to measure the RAM memory is a MB, we can state that a deployment component provides 2GB of RAM simply by associating to a given deployment component $1024 * 2$ units of memory. When two objects consume an amount r_1 and r_2 of the same resource we assume that the cumulative consumption does not exceed the sum of r_1 and r_2. Obviously, a resource can never be consumed in more quantities than provided.

We require an annotation for every relevant class that can be involved in the automatic generation of the main.For instance, there is no need to annotate a class implementing an internal data structure. 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`.

The ANTLR[7] grammar of the annotation language is reported in Table 2 and a specific example of annotated ABS code is in Listing 1 (annotations in Lines 4–8).

[7] http://www.antlr.org

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

Table 2: Grammar of ABS annotations

```
1  interface IQueryService extends Service {
2     List<Item> doQuery(String q);
3  }
4  [Deploy: scenario[
5     MaxUse(1),
6     Cost("CPU", 1), Cost("Memory", 400),
7     Param("c", Default("CustomerX"),
8     Param("ds", Req)]]
9  class QueryServiceImpl(DeploymentService ds,
10   Customer c) implements IQueryService {
11    // Implementation
12 }
```

Listing 1: Fredhopper Query API

In general, 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`.
- `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.

We now discuss the annotated ABS code of Listing 1 taken from the specification of the Query API of Fredhopper Cloud Services described in Section 3.

Abstracting away the implementation details, the Query API has been modelled as a `QueryServiceImpl` class implementing the interface `IQueryService`. The interface and the class `QueryServiceImpl` are defined in ABS at Lines 3 and 9. The annotation for the class `QueryServiceImpl` is introduced before the class definition, at Line 4. The annotation at Line 5 specifies that an object of `QueryServiceImpl` may be used as parameter only once during the creation of other objects. Line 6 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 7 and 8 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 the string `CustomerX`.

As mentioned above, 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.

```
1 [Deploy: scenario[ Name( "NewCustomer")
2    MaxUse(1),
3    Cost("CPU", 2), Cost("Memory", 200),
4    Param("c", Default("NewCustomer"),
5    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.[8]

5. 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).

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

Table 3: 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 3, a constraint in DDLang 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 19), (iii) the number of objects of a class `C` (denoted `CLASS[C]` - Line 20). 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 21).

With this expressivity 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
```

More complex quantities are concerned with deployment components. These are expressed (Line 6) 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

[8]Please note the annotation in Listing 1 represents the default scenario (`Def`) since the `Name` annotation is not defined.

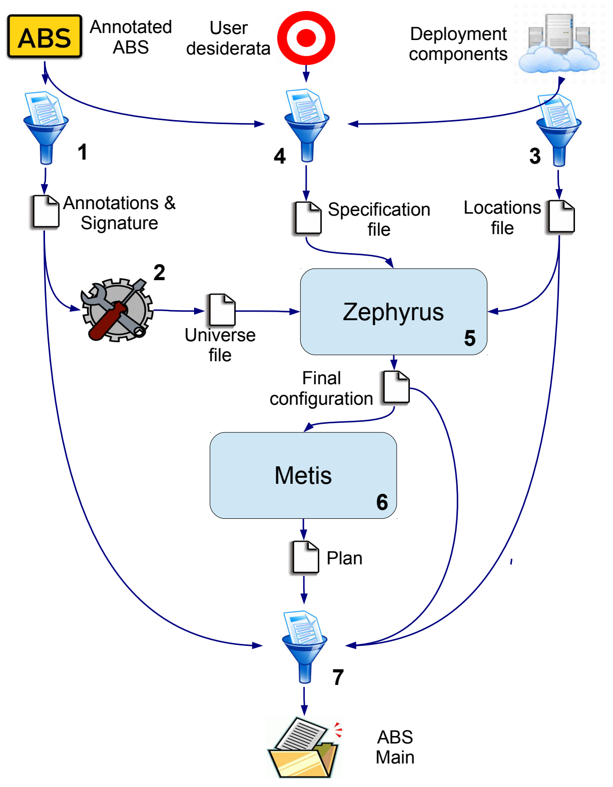

Figure 3: MODDE execution flow

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
```

The impossibility to co-locate two objects in the same deployment component can be expressed in a similar manner. For example, in our case study, we require that `PlatformServiceImpl` and `LoadBalancerServiceImpl` are installed separately for fault tolerance reasons. This requirement is captured by the following constraint.

```
DC[ CLASS[PlatformServiceImpl] > 0 and
    CLASS[LoadBalancerServiceImpl] > 0 ] = 0
```

6. 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 following the workflow depicted in Figure 3. More precisely, MODDE takes three distinct inputs: the ABS program annotated as discussed in Section 4, the user desiderata formalized as constraints in the language DDLang defined in Section 5, 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 (monetary) 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",
    "provide_resources" :
      {"CPU" : 2, "Memory" : 375},
    "cost" : 105 },
  { "name" : "c3.xlarge",
    "provide_resources" :
      {"CPU" : 4, "Memory" : 750},
    "cost" : 210 } ],
 "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 a parse-tree of the annotated ABS program, retrieving all the annotations and the class signatures. This step (step 1 in Figure 3) 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. Indeed, Zephyrus requires as input a representation of the components to deploy following the Aeolus model specification [7]. 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. Active require ports must be bound to active provide ports of other components.

We model an ABS object `obj` 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, in our use case, 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 class `QueryServiceImpl` requires the functionality `DeploymentService` in the `On` state. Dually, since the class `QueryServiceImpl` implements the

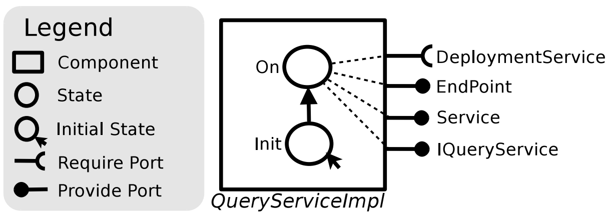

Figure 4: Aeolus model representation for object of class `QueryServiceImpl`

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 4.

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. [9]

The first input of the Zephyrus tool is the universe of all the components obtained from the annotated classes. Moreover, to compute 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 3) 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. This translation is rather straightforward since the specification of Zephyrus is more expressive than DDLang.[10]

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 [13]. Even though this theoretical complexity is quite high, in practice in our tested scenarios (but also in other deployment scenarios such as those discussed in [8]) Zephyrus was able to successfully compute the optimal solution in reasonable time (i.e., few minutes or less).

Since Zephyrus can be used to minimize different quantities we use it to minimize the total cost of all the deployment components used. 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 that need to be installed. To get this information, in step 6, we launch Metis. 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`.

7. VALIDATION

In order to validate 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 AWS instance. We defined the capacity of each resource for several AWS instance types in the locations file.[11] The price used in the cost attribute of each AWS instance type concerns on-demand instances in the US East region running Linux.[12]

We created several deployment scenarii 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 (and this varies over time: visitor peaks are typically observed around Christ-

[9]In ABS a single class `C` can expose several interfaces (see, e.g., the three interfaces in the provide ports of Figure 4). In this case, the `MaxUse(n)` indicates the maximal usage of the object of class `C` for every single provided interface.

[10]In the Zephyrus specification it is indeed possible to have also global variables and additional constraint on the locations. For more details about the specification language of Zephyrus we refer the interested reader to [6].

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

[12]`http://aws.amazon.com/ec2/pricing/`

mas or during promotions). 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. Below we list some of these requirements.

- Platform Service and Service Provider should be co-located, but no other Services should reside at the same location, and there is only one instance of Platform Service (shared by all customers).
- Load Balancer should not be co-located with other Services and is dedicated per customer (for large customers, there may be multiple Load Balancers).
- Query Service should always be deployed together with the Deployment Service on a dedicated machine (per customer).

Section 5 shows the formal versions of some of the above requirements. The specification language proved to be sufficiently expressive to capture the above and all other requirements.

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 modelling of the framework, 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) and is used to deploy the object implementing the Amazon interface.

We have automatically generated ABS deployments for several scenarii. We report and comment only the result obtained by MODDE when 2 instances of the Query service are required for a customer,[13] which is a simple but already significative 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)]);
```

[13]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.

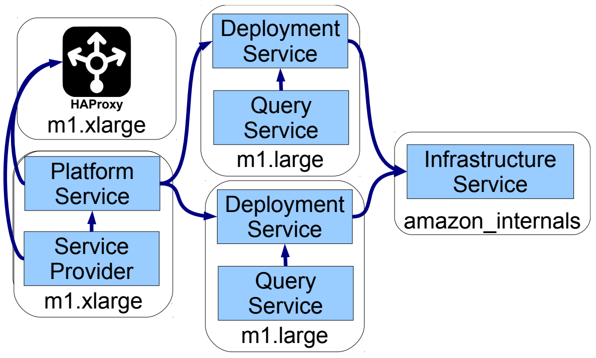

Figure 5: Example of automatic objects allocation to deployment components.

```
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 5. Deployment components are depicted as boxes containing 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 costumers that have already `m1` running machines. New costumers 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 extremely important 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.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a new way to tackle and unify the modelling of a software 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 modelling language, but we are not limited to it: other languages that have primitives to handle the deployment aspects can be used as well, provided that annotation related to the execution costs of the system are used. The desiderata of the final system are then specified in the form of constraints written in a domain specific language. These constraints are processed by MODDE. The result is an automatically generated main program that deploys the system and satisfies the user wishes.

MODDE has been validated on an industrial case study from the e-Commerce company Fredhopper. The results are encouraging since the deployment solutions generated by MODDE resemble those devised by the operations. This is a complex, time consuming process that requires in-depth domain specific knowledge. Clearly, any automated tool that can give quicker and better evaluations of the deployment configuration based on a rigorous formal approach is a big step forward.

This is just the first step towards the possibility of having a one button click deployment solution. Indeed, ideally, the annotations that the developer now has to enter manually, could be inferred automatically using formal methods tools such as [3]. Unfortunately, these techniques are not yet sufficiently mature to be used in a production environment. For this reason we currently resort to manual annotations.

As of today, our approach simply consider systems whose configuration is obtained by using only the initialization parameters. In more complex cases, the configuration should involve also method indications to pass configuration information to objects also after their creation. For instance, this is necessary in cases in which there are mutual or circular dependencies, where multi-stage configuration is usually adopted [2]. As a future work, we will consider annotations on method signatures in order to be able to automatically compute deployments involving also method invocations.

Based on the feedback from the operations team at Fredhopper, we would also like to improve some functionalities of MODDE. For instance, we would like to find the best deployment configuration given a user-specified maximal cost and a maximal resource consumption. Furthermore, annotations could be enriched with parametric costs that depend on the class parameters. The declarative language can be simplified by adding "syntactic sugar" that allows users to enter their desiderata in a more concise and readable way. Moreover, we would also like to tackle the computational aspects involved in the process of finding the optimal configuration. Even though it did not happen during our validation, the optimization problem, being NP-hard in nature, may take a lot of time to be solved. For this reason, we would also like to exploit heuristics such as local search techniques that quickly provide good solutions, even though they are suboptimal or not provably optimal.

9. REFERENCES

[1] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. MPM: a modular package manager. In *CBSE'11: 14th symposium on component based software eng.*, 2011.

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

[3] E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, and P. Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications*, 8(4):323–339, 2014.

[4] Apache Software Foundation. Apache Brooklyn. https://brooklyn.incubator.apache.org/.

[5] M. Burgess. A Site Configuration Engine. *Computing Systems*, 8(2), 1995.

[6] 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.

[7] R. D. Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239, 2014.

[8] R. Di Cosmo, A. Eiche, J. Mauro, G. Zavattaro, S. Zacchiroli, and J. Zwolakowski. Automatic Deployment of Software Components in the Cloud with the Aeolus Blender. Technical report, Inria Sophia Antipolis, 2015.

[9] X. Etchevers, T. Coupaye, F. Boyer, and N. D. Palma. Self-Configuration of Distributed Applications in the Cloud. In *CLOUD*, 2011.

[10] I. Feinerer. Efficient large-scale configuration via integer linear programming. *AI EDAM*, 27(1):37–49, 2013.

[11] 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.

[12] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, 2012.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[14] 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.

[15] 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.

[16] HashiCorp. Terraform. https://terraform.io/.

[17] J. A. Hewson, P. Anderson, and A. D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, 2012.

[18] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[19] Juju, devops distilled. https://juju.ubuntu.com/.

[20] L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1), 2006.

[21] T. A. Lascu, J. Mauro, and G. Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *ICTAI*, 2013.

[22] T. A. Lascu, J. Mauro, and G. Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *FACS*, 2013.

[23] J. Mirkovic, T. Faber, P. Hsieh, G. Malaiyandisamy, and R. Malaviya. DADL: Distributed Application Description Language. *USC/ISI Technical Report*, 2010.

[24] OASIS. Cloud Application Management for Platforms. http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.

[25] 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.

[26] Opscode. Chef. http://www.opscode.com/chef/.

[27] Puppet Labs. Marionette collective. http://docs.puppetlabs.com/mcollective/.

[28] K. Schmid and A. Rummler. Cloud-based software product lines. In *SPLC*, 2012.

[29] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

Appendix B

Deployment Variability in Delta-Oriented Models

Deployment Variability in Delta-Oriented Models [⋆]

Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
`{einarj,rudi,sltarifa}@ifi.uio.no`

Abstract. Software engineering increasingly emphasizes variability by developing families of products for a range of application contexts or user requirements. ABS is a modeling language which supports variability in the formal modeling of software by using feature selection to transform a delta-oriented base model into a concrete product model. ABS also supports deployment models, with a separation of concerns between execution cost and server capacity. This allows the model-based assessment of deployment choices on a product's quality of service. This paper combines deployment models with the variability concepts of ABS, to model deployment choices as features when designing a family of products.

1 Introduction

Variability is prevalent in modern software in order to satisfy a range of application contexts or user requirements [34]. A software product line (SPL) realizes this variability through a family of product variants (e.g., [29]). A specific product is obtained by selecting features from a feature model [36]; these models typically focus on the functionality and software quality attributes of different features and products. To express variability in system *design*, features typically take the form of architectural models, behavioral models, and test suites [35]. Architectural variability [16] focuses on the presence of component variants, and can be described using, e.g., the Variability Modeling Language [27], UML stereotypes [14], or (hierarchical) component models such as Koala [37]. In Delta modeling [10, 30, 31], a set of deltas specifies modifications to a core product. Δ-MontiArch applies delta modeling to architectural description [15]; a delta can add or remove components, ports, and connections between components.

Whereas architectural models describe the *logical* organization of a system in terms of components and their connections, we are interested in the *physical* organization of software units on physical or virtual machines; we call this physical organization the *deployment architecture*. Varying deployment architectures will perform the same computations, but with different cost and/or time spent. Thus, a deployment architecture comprises specifications of execution costs and available resources.

[⋆] Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`).

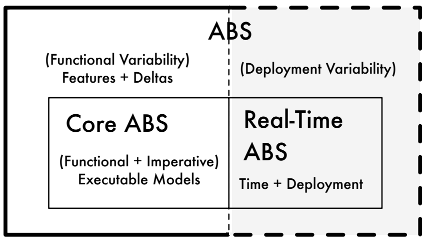

Fig. 1. ABS language extension.

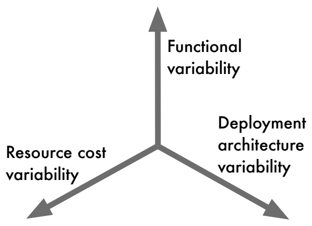
Functional variability
Deployment architecture variability
Resource cost variability

Fig. 2. The SPL variability space with deployment variability.

This paper integrates deployment variability in SPL models such that different targeted deployment architectures may be taken into account early in the design of the SPL. We aim at a reasonable orthogonality between functional and deployment variability in the SPL model. The starting point for this work is the **a**bstract **b**ehavioral **s**pecification language ABS, which adds support for variability to models in the kernel modeling language Core ABS [20]. ABS is *object-oriented* to be easy to use for software developers; it is *executable* to support code generation and (timed) validation of models; and it has a *formal semantics* which enables the static analysis of models (e.g., the worst-case resource consumption can be derived for a model). ABS is particularly suitable for our objective because (1) ABS supports SPL modeling based on deltas [9, 11], and (2) ABS supports the modeling of deployment decisions based on the modeling concept of *deployment components* [23] in Real-Time ABS [7]. Real-Time ABS leverages resources and their dynamic management to the abstraction level of software models. Fig. 1 shows how functional variability modeling in ABS and time and deployment models in Real-Time ABS both extend Core ABS. Although these extensions of ABS coexist, they have so far never been combined. The purpose of this paper is to combine these two extensions in order to model deployment variability, corresponding to the dotted area in Fig. 1.

Our approach to deployment variability for SPL models makes a separation of concerns between cost and capacity which introduces two new variation points in the variability space of ABS feature models (depicted in Fig. 2):

– **Resource cost variability**: These features determine the costs associated with executing the SPL's logical artifacts; and
– **Deployment architecture variability**: These features determine how the logical artifacts are deployed on locations with different execution capacities.

The main contribution of the paper is an integration of delta models with deployment architectures in ABS. This integration allows orthogonality between functional and deployment variability, such that features expressing functionality, resource cost and deployment variability are kept in different trees in the ABS feature models. The integration is illustrated by variability patterns for MapReduce [12], a programming model for highly parallelizable programs. Furthermore, this integration allows ABS tools to be used to analyze functional features with respect to deployment architecture during the early design stage of SPLs.

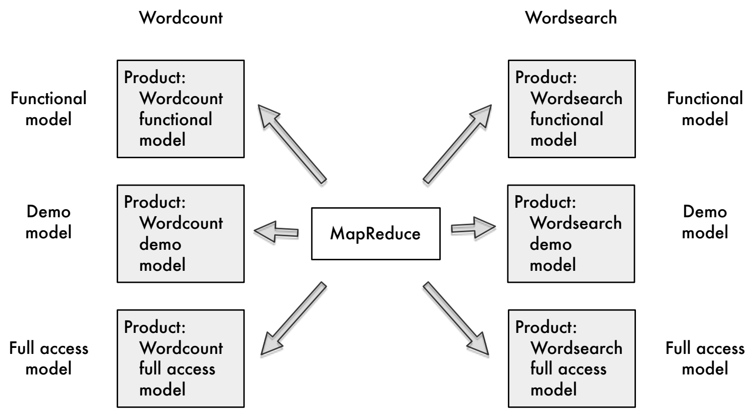
Wordcount
Wordsearch
Functional model
Product: Wordcount functional model
Product: Wordsearch functional model
Functional model
Demo model
Product: Wordcount demo model
MapReduce
Product: Wordsearch demo model
Demo model
Full access model
Product: Wordcount full access model
Product: Wordsearch full access model
Full access model

Fig. 3. A family of products sharing an underlying MapReduce structure.

Paper overview. Sect. 2 motivates our work by an example of deployment variability. Sect. 3 presents modeling in the abstract behavioral specification language ABS and Sect. 4 delta modeling and its realization in ABS. Sect. 5 combines delta-oriented variability with deployment modeling, and discusses how to extend a feature model with deployment variability. Sect. 6 revisits the example, Sect. 7 discusses related work, and Sect. 8 concludes the paper.

2 Motivating Example

MapReduce [12] is a programming pattern for processing large data sets in two stages; first the *Map* stage separates parallelizable jobs on distinct subsets of data to produce intermediate results, then the *Reduce* stage merges the intermediate data into a final result. The initial and intermediate data are on the form of key/value pairs, and the final result is a list of values per key. MapReduce does not specify the computations done by the two stages or the distribution of workloads across machines, making it a good abstract base model for SPLs.

Our example uses MapReduce to model product variants of a range of services which inspect a set of documents. Individual products may implement, e.g., Wordcount, which counts the occurrences of words in the given documents, and Wordsearch, which searches for documents in which a given word occurs. For simplicity, we assume that a service either provides the Wordcount or the Wordsearch feature. The services are implemented on a cluster of computers, using MapReduce.

To attract clients to the word count and word search services, freely available demo versions offer the same functionality as the full versions, albeit with a lower quality of service. When the services are deployed, the demo versions will run on a few machines, whereas the full versions have access to the full power of the cluster. Our model has three versions of each service: the purely functional model, the model with full access to the cluster, and a model with restricted access to the cluster. This product family (see Fig. 3) is a running example in the paper.

3 Behavioral and Deployment Modeling in ABS

The abstract behavioral specification language ABS targets the executable design of distributed object-oriented systems. It has a formally defined kernel called Core ABS [20]. ABS is based on concurrent object groups (COGs), akin to concurrent objects [8,21], Actors [1], and Erlang processes [5]. COGs support interleaved concurrency based on guarded commands. ABS has a functional and an imperative layer, combined with a Java-like syntax. Real-Time ABS [7] extends Core ABS models with (dense) time; in this paper we do not specify execution time directly but rather *observe* time by measurements of the executing model.

ABS has a *functional layer* with algebraic data types such as the empty type `Unit`, booleans `Bool`, integers `Int`; parametric data types such as sets `Set<A>` and maps `Map<A, B>` (for type parameters `A` and `B`); and functions over values of these data types, with support for pattern matching. The modeler can define additional types to succinctly express data structures of the problem domain.

The *imperative layer* of ABS describes side-effectful computation, concurrency, communication and synchronization. ABS objects are *active* in the sense that their `run` method, if defined, gets called upon creation. Communication and synchronization are decoupled: Communication is based on asynchronous method calls. After executing `f=o!m(e)`, which assigns the call to a *future variable* `f`, the caller proceeds execution *without blocking* while `m(e)` executes in the context of `o`. Two operations on future variables control synchronization in ABS. First, the statement **await** `f?` *suspends the active process* unless a return value from the call associated with `f` has arrived, allowing other processes in the same COG to execute. Second, the return value is retrieved by the expression `f.`**get**, which *blocks all execution in the COG* until the return value is available. Inside a COG, Core ABS also supports standard synchronous method calls `o.m(e)`.

A COG can have at most one active process, executing in one of the objects of the COG. Scheduling is cooperative via **await** `g` statements, which suspend the current process until `g` (a condition over object or future variable state) becomes true. The remaining statements of ABS (assignment, object creation, conditionals and loops) are designed to be familiar to a Java programmer.

Deployment Modeling. One purpose of describing deployment in a modeling language is to differentiate execution time based on *where* the execution takes place, i.e., the model should express how the execution time varies with the available *capacity* of the chosen deployment architecture. For this purpose, Real-Time ABS extends Core ABS with primitives to describe *deployment architectures* which express how distributed systems are mapped on physical and/or virtual media with many locations. Real-Time ABS lifts deployment architectures to the abstraction level of the modeling language, where the physical or virtual media are represented by *deployment components* [22].

A *deployment component* is part of the model's deployment architecture, on which a number of COGs are deployed. Deployment components are first-class citizens and they support a number of methods for load monitoring and load balancing purposes (cf. [22]). Each deployment component has an *execution capacity*, which is the amount of resources available per accounting period. By default, all objects execute in a default (root) environment with unrestricted capacity. Other deployment components with restricted capacities may be created to capture different deployment architectures. COGs are created on the same deployment component as their creator by default; a different deployment component may be selected by an optional *deployment annotation* [DC: dc] to object creation, for a deployment component dc.

The available resource capacity of a deployment component determines the amount of computation which may occur in the objects deployed on that deployment component. Objects allocated to the deployment component compete for the shared resources in order to execute, and they may execute until the deployment component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the deployment component define its capacity inside an accounting period, after which the resources are renewed.

The resource consumption of executing statements in the Real-Time ABS model is expressed by means of adding a *cost annotation* [Cost: e] to any statement. It is the responsibility of the modeler to specify appropriate resource costs. A behavioral model may be gradually transformed to provide more realistic resource-sensitive behavior by inserting more fine-grained cost annotations. The automated static analysis tool COSTABS [2] can compute a worst-case approximation of resource consumption, based on static analysis techniques. However, the modeler may also want to capture *normative* constraints on resource consumption, such as resource limitations, at an abstract level; these can be made explicit in the model during the very early stages of the system design. To this end, cost annotations may be used by the modeler to abstractly represent the cost of some computation which is not fully specified in the model.

4 Delta-Oriented Variability in ABS

This section describes how SPLs are modeled in ABS. ABS includes a delta-oriented framework for variability [9,11]. Fig. 4 depicts a delta-oriented variability model where a feature model F with orthogonal variability [18] is represented as two trees that hierarchically structure the set of features of this model. Sets of features from the feature model F are linked to sets of delta modifications from the delta model Δ, which apply to the common base model P to produce different product line configurations C, C' and C', and finally a specific product ρ is extracted from the product line configuration C.

Feature model. A feature model in ABS is represented textually as a forest of nested features where each tree structures the hierarchical dependencies between related features, and each feature in a tree may have a collection of Boolean or integer attributes. The ABS feature model can also express other

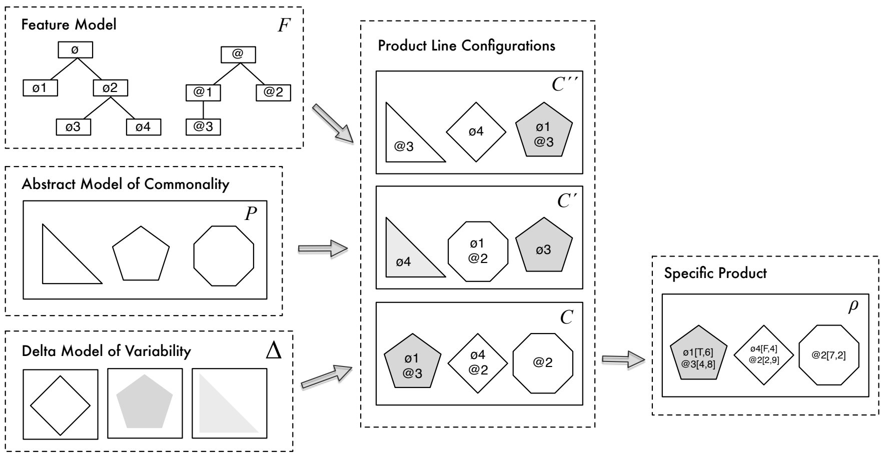

Fig. 4. A graphical representation of a Delta-Oriented variability model.

cross-tree dependencies, such as mandatory and optional sub-features, and mutually exclusive features. The **group** keyword is used to specify the sub-features of a feature; the **oneof** keyword means that exactly one of the sub-features must be selected in the created product line, the range of values associated to an attribute specify the values in which an attribute can be instantiated when an specify product is generated. For the full details, we refer the reader to [9, 11].

Example 1. In the functional feature model of the MapReduce example from Section 2, a tree with a root Calculations offers two alternative and mutually exclusive features that can be selected to express that a specific product supports counting words or searching for words.

```
root Calculations { group oneof { Wordcount, Wordsearch }}
```

In addition ABS allows a feature model with multiple roots (hence, multiple trees) to describe orthogonal variability [18], which is useful for expressing unrelated functional and other features (e.g., features related to quality of service).

Delta model. The concept of delta modeling was introduced by Schaefer et al. [6, 31–33] as a modeling and programming language approach for SPLs. This approach aims at automatically generating software products for a given valid collection of features, providing flexible and modular techniques to build different products that share functionality or code. In delta-oriented programming, application conditions over the set of features and their attributes, are associated with units of program modifications called delta modules. These delta modules may add, remove, or otherwise modify code. The implementation of an SPL in delta-oriented programming is divided into a common core module and a set of delta modules. The core module consists of classes that implement a complete product of the SPL. Delta modules describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product.

Technically, delta modules have a unique identifier, a list of parameters, and a body containing a sequence of class and interface modifiers. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The modifications can occur within a class or interface body, and modifier code can refer to the original method by using the **original**() keyword. Delta modules in ABS can be parametrized by attribute values to enable the application of a single delta in more than one context.

Product line configuration. The product line configuration links feature models with delta modules to provide a complete specification of the variability in an ABS product line. A product line configuration consists of the set of features of the product line and a set of delta clauses. Each delta clause names a delta module and specifies the conditions required for its application, called application conditions. A partial ordering on delta modules constrains the order in which delta modules can be applied to the core module.

Specific product. A product selection clause generates a specific product from an ABS product line. It states which features are to be included in the product and specifies concrete values for their attributes. A product selection is checked against the feature model for validity. The product selection clause is used by the product line configuration to guide the application of the delta modules during the generation of the final product.

Generated final product. Given a Core ABS program P, a set of delta modules Δ, a product line configuration C, and a feature model F (as depicted in Fig. 4), the final product ρ, which will be a Core ABS program, is derived as follows: First check that the selection of features for ρ satisfies the constraints imposed by the feature model F; then select the delta modules from Δ with a valid application condition with respect to ρ; and finally apply the delta modules to the core program P in some order respecting the partial order described in C, replacing delta parameters in the code with the literal values supplied by the feature.

5 Deployment Variability in ABS

Feature models usually describe functional variability in a software product line. This section discusses lifting deployment variability to ABS feature models and its interaction with functional variability. Our approach aims to establish orthogonality between the functional and deployment aspects in an SPL model in order to maintain multiple axes of variability (see Fig. 2). The further separation of concerns between cost and capacity in the deployment models of ABS is reflected in the feature models as well.

Thus, variability in a deployment-aware SPL comprises these variation points in the feature models:

Functional variability: These features determine the functional behavior of a product and are used as in standard SPL engineering.

Resource cost variability: These features describe the choice of how the incurred resource cost is estimated during execution of the model. The basic feature is the *no cost* feature, typically selected for functional analysis of the SPL model. Other cost models are fixed-cost for selected jobs (similar to costs in a basic queuing network or simulation model; see, e.g., [19]), and data-sensitive costs. These can be either measured, real cost for selected jobs or worst-case approximations (which may depend on data flow as well as control flow). All of these can be expressed via cost annotations.

Deployment architecture variability: These features determine how the logical artifacts of the model are mapped to a specific deployment architecture, which determines the execution capacity of the different locations on which the logical artefacts execute. The basic feature is the *undeployed* feature which does not impose any capacity restrictions on the execution. This feature is typically selected together with *no cost* during functional analysis and testing. When analyzing non-functional properties, features describe how selected parts of the logical architecture are deployed on deployment components with restricted capacity, either statically or (for virtualized deployment) dynamically.

Example 2. We extend the feature model of Example 1 with a Resources tree for resource costs, and a Deployments tree for deployment architecture. The Resources root has the basic feature NoCost, the feature FixedCost for a basic data-independent cost model specified in the attribute cost, the feature WorstcaseCost for a worst-case cost model in terms of the size of the input files, and MeasuredCost for using the actual incurred cost measured during execution of the model. The Deployments root has three alternative features related to the number of available machines in the physical deployment architecture; the capacity of each machine is specified by the attribute capacity.

```
root Resources {
  group oneof {
    NoCost,
    FixedCost { Int cost in [ 0 .. 10000 ] ; },
    WorstcaseCost,
    MeasuredCost
  }
}
root Deployments {
  group oneof {
    NoDeploymentScenario,
    UnlimitedMachines { Int capacity in [ 0 .. 10000 ] ; },
    LimitedMachines { Int capacity in [ 0 .. 10000 ] ; Int machinelimit in [ 0 .. 100 ] ; }
  }
}
```

```
// These definitions to be changed in delta modifications
type InKeyType = String; // filename
type InValueType = List<String>; // file contents
type OutKeyType = String; // word
type OutValueType = Int; // count

interface MapReduce {
  List<Pair<OutKeyType, List<OutValueType>>>
      mapReduce(List<Pair<InKeyType, InValueType>> docs); // invoked by client
  Unit finished(Worker w); // invoked by workers when finished with 1 task
}

interface IMap { // invoked by MapReduce controller
  List<Pair<OutKeyType, OutValueType>>
    invokeMap(InKeyType key, InValueType value);
}

interface IReduce { // invoked by MapReduce controller
  List<OutValueType>
    invokeReduce(OutKeyType key, List<OutValueType> value);
}

interface Worker extends IMap, IReduce { }
```

Fig. 5. Interfaces of the base model of the MapReduce example in ABS.

6 Example: Product Variability in the MapReduce Example

This section describes the implementation of a generic MapReduce framework in ABS and its adaptation to different products as described in Section 2. It will become apparent that a product that is implemented according to best practices for object-oriented software (i.e., decomposing functionality, methods implementing one task only, and the careful definition of datatypes) also makes the product well-suited as a base product for a software product line.

6.1 Commonalities in the ABS Base Product

Fig. 5 shows the interfaces for the main MapReduce object and for the Worker objects which will carry out the computations in parallel. The computation is started by calling the mapReduce method with a list of *(key, value)* pairs. The main object will then create a number of worker objects, call invokeMap on these objects, gather and collate the results of the mapping phase, call invokeReduce on the workers and collate and return the final result.

The base product in our example implements a word count function (computing word occurrences over a list of files), without a resource or deployment model. Worker objects are reused from a pool, but there is no bound on the number of workers created. Workers add themselves back to the pool by calling finished.

Figure 6 shows part of the worker implementation of the base product (i.e., a Wordcount product without any cost model). The invokeReduce method sets up the result, calls a private method reduce which emits intermediate results using

```
class Worker(MapReduce master) implements Worker {
  List<OutValueType> reduceResults = Nil;

  List<OutValueType> invokeReduce(OutKeyType key, List<OutValueType> value) {
    reduceResults = Nil;
    this.reduce(key, value);
    List<OutValueType> result = reduceResults;
    reduceResults = Nil;
    master!finished(this);
    return result;
  }

  Unit emitReduceResult(OutValueType value) {
    reduceResults = Cons(value, reduceResults);
  }

  // variation point for functional model
  Unit reduce(OutKeyType key, List<OutValueType> value) {
    OutValueType result = 0;
    ... // sum up value list into result variable ...
    this.emitReduceResult(result);
  }
}
```

Fig. 6. The reduce part of the Wordcount example in the Worker class.

the method emitReduceResult. The reduce method in Fig. 6 is equivalent to the one shown in the original MapReduce paper [12]. The mapping functions of the worker objects are implemented in the same way.

6.2 Variability in the ABS Product Line

To change the functional feature of the model from computing word counts to computing word search, some parts of the model need to be altered via delta application. The same applies when varying the deployment and cost model, as explained in Section 5. These variation points turn out to be orthogonal and can be modified independently of each other.

In the example, the methods to be modified by deltas are not public; i.e., they are not part of the published interface of the classes comprising the base model. This appears to be a recurring pattern: public methods like invokeReduce of Fig. 6 interact with the outside world, gather and decompose data for computation and returning. If the modeler factors out computation into private methods with only one single task to perform (like reduce in Fig. 6), these methods can be cleanly replaced in deltas, without imposing constraints on the implementation. This suggests that clean object-oriented code will in general be likely to be amenable to delta-oriented modification.

Functional variability. The following delta shows a delta fragment that modifies the functionality of the base model:

```
delta DOccurrences;
modifies type OutValueType = String; // Change the method signatures
modifies class Worker {
  modifies Unit map(InKeyType key, InValueType value) {
    ... // change non−public map method to compute occurrences
  }
  modifies Unit reduce(OutKeyType key, List<OutValueType> value) {
    ... // change non−public reduce method to compute occurrences
  }
}
```

By modifying the type synonyms InKeyType, InValueType, OutKeyType and OutValueType from the base model, we can change the data types and method signatures of the model without having to change any code in the MapReduce class. Modifying the methods map and reduce of the Worker class changes the computation performed by the product. The new map and reduce methods use emitMapResult and emitReduceResult as in the base model; hence they do not need to care about invocation or return value handling protocols.

Resource cost variability. Costs are incurred during (and because of) computational activity. This means that cost model and functional model are related. However, the two aspects can be decoupled

cleanly via the **original**() call, which we use to associate the given cost with the original code. Care must be taken in the productline definition to ensure that any deltas incurring costs are applied *after* deltas modifying functionality; otherwise, the cost association would be overwritten.

```
delta DFixedCost (Int cost);
modifies class Worker {
  modifies Unit emitMapResult(OutKeyType key, OutValueType value) {
    [Cost: cost] original(key, value);
  }
  modifies Unit emitReduceResult(OutValueType value) {
    [Cost: cost] original(value);
  }
}
```

This FixedCost delta assigns a cost (given as a delta attribute) to each computation of an intermediate result; the feature attribute is passed in as a delta parameter. In general, costs are introduced into MapReduce by wrapping the methods invokeMap and invokeReduce for assigning costs to starting a computation step, and by modifying emitMapresult and emitReduceresult for assigning costs to the production of a result. Figure 6 shows where these methods are invoked.

An alternative approach to adding resource costs via hooks is to use the ABS **original**() call, wrapping the original map, emitMapresult etc. methods with costs. This approach makes the functional model simpler, but leads to a more complicated product line configuration since the correct order of delta application must be specified in that case.

Deployment architecture variability. Deployment architecture, i.e., decisions on how many workers to create and how many resources to supply them with, is implemented in the MapReduce class. As mentioned, this class manages a pool of Worker instances which is by default of unbounded size. To change this behavior, the modeler implements a delta that overrides a method getWorker (and also the method finished of the MapReduce implementation in case the new

```
productline MapReduceSPL;

features
  Wordcount, Wordsearch,                                    // Functional features
  NoCost, FixedCost, WorstCaseCost, MeasuredCost,           // Resource cost features
  NoDeploymentScenario, UnlimitedMachines, LimitedMachines; // Deployment architectures

delta DOccurrences when Wordsearch;
delta DFixedCost(Cost.cost) after DOccurrences when Cost;
delta DUnboundedDeployment(UnlimitedMachines.capacity) when UnlimitedMachines;
delta DBoundedDeployment(LimitedMachines.capacity, LimitedMachines.machinelimit)
      when LimitedMachines;
...
```

Fig. 7. Product line configuration for the MapReduce example in ABS.

```
product WordcountModel (Wordcount, NoCost, NoDeploymentScenario);
product WordcountFull (Wordcount, Cost{cost=10}, UnlimitedMachines{capacity=20});
product WordcountDemo (Wordcount, Cost{cost=10},
    LimitedMachines{capacity=20, machinelimit=2});

product WordsearchModel (Wordsearch, NoCost, NoDeploymentScenario);
product WordsearchFull (Wordsearch, Cost{cost=10}, UnlimitedMachines{capacity=20});
product WordsearchDemo (Wordsearch, Cost{cost=10},
    LimitedMachines{capacity=20, machinelimit=2});
```

Fig. 8. Specifying Products for the MapReduce example in ABS.

getWorker method does not use the resource pool of the base model). The capacity and number of deployment components can be adjusted via delta parameters:

```
delta DBoundedDeployment (Int capacity, Int maxWorkers);
modifies class MapReduce {
  ... // adjust behavior of resource pool and capacities of created deployment components
}
```

The product line configuration. The feature model presented in Section 5 extends the model of Section 2 with resource cost variability, resulting in 14 different products. Fig. 7 shows part of the product line configuration and Fig. 8 shows the specification of some of the derivable products.

In the deployment components of the deployment architecture features, capacity is defined by the amount of resource costs that can be processed per accounting period (in terms of the dense time semantics of execution in Real-Time ABS). When the base model is extended with features for deployment architecture and resource cost, the *load* on the individual deployment components, defined as the actual incurred cost per accounting period, can be recorded and visualized.

We illustrate how deployment variability for products can be validated using the simulation tool of ABS, by comparing the performance of two different deployments of the Wordcount product, varying the number of available machines between 5 (the "Demo" version) and 20 (the "Full" version), but keeping the cost model, input data and computation model constant. The graphs in Fig. 9 shows the total load of all machines over simulated time for the two products. The figure shows two typical instances of a typical MapReduce workload; first, the

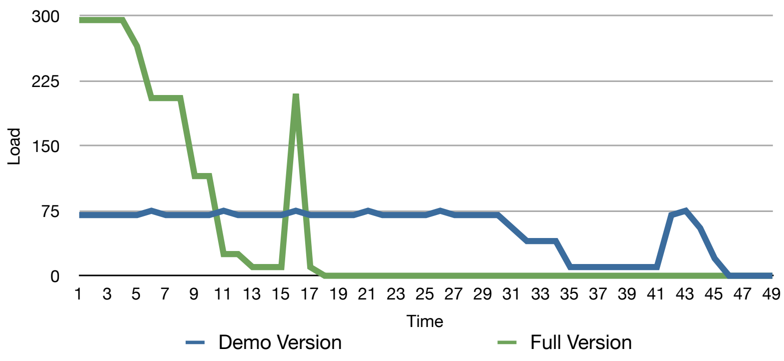

Fig. 9. Varying deployment model, constant cost and functional model

map processes execute until they are finished, then the reduce processes execute. The start of the reduce phase can be observed in the graph of Fig. 9 as the second spike in processing activity. It can be seen that the demo version takes over twice as much simulated time to complete its execution, while the full version completes its execution earlier by incurring a load that is higher than for the demo version (while still decreasing as the map processes terminate).

Similar qualitative investigations can be performed regarding the influence of varying cost models (e.g., worst-case vs. average cost) and more involved deployment strategies.

7 Related Work

The inherent compositionality of the concurrency model considered in this paper allows objects to be naturally distributed on different locations, because only an object's local state is needed to execute its methods. In previous work [4, 22, 23], the authors have introduced *deployment components* as a modeling concept to captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources. The formal details of this approach are given in [23]; two larger case studies on virtualized systems deployed on the cloud are presented in [3, 24]. Our approach to deployment modeling would be a natural fit for resource-sensitive deployment in other Actor-based approaches, e.g., [5, 17].

Deployment variability is not considered in the recent software diversity survey [35], but it has been studied in the context of feature models. For example, a feature model that captures the architectural and technological variability of multilayer applications is described in [13] together with an associated model-driven development process. In contrast our paper considers a much simpler feature model, but it is integrated in a full SPL framework and explicitly linked to executable models which can be compared by tool-based analysis. Without considering variability, a platform ontology and modeling framework based on description logic is proposed by [38], which can be used to automatically configure various reusable concrete platforms that can be later be integrated with a platform-independent model using the Model Driven Architecture approach. We follow a similar approach based on the extending a purely functional model with deployment features, but our framework is based on simpler concepts which does not introduce the overhead of description logic. In the context of QoS variability, [25] study a modeling and analysis framework for testing the QoS of an orchestration before deployment to determine realistic Service Level Agreement contracts; their analysis uses probabilistic model of QoS. Our work similarly allows the model-based comparison of QoS variability, but focuses on deployment architecture and processing capacity rather than orchestration.

The MapReduce programming pattern which is the basis for the example of this paper, has been formalized and studied from different perspectives. [39] develop a CSP model of MapReduce, with a focus on the correctness of the communication between the processes. [26] develops a rigorous description of MapReduce using Haskell, resulting in an executable specification of MapReduce. [28] formalizes an abstract model of MapReduce using the proof assistant Coq, and use this formalization to verify JML annotations of MapReduce applications. However, none of these works focus on deployment strategies or relate MapReduce to deployment variability in SPLs.

8 Conclusion

Software today is increasingly often developed as a range of products for devices with restricted resource capacity or for virtualized utility computing. For an SPL targeting such platforms, the deployment of different products in the range should also be considered as a variation point in the SPL.

This paper integrates explicit resource restricted deployment scenarios into a formal modeling language for SPL engineering. This integration is based on delta models to systematize the derivation of product variants, and demonstrated in the ABS modeling language. The proposed integration emphasizes orthogonality between functional features, resource cost features, and deployment architecture features, to facilitate finding the best match between functional features and a target deployment architecture for a specific product. The supported analysis allows the validation of deployment decisions for specific products in the SPL, which may entail a refinement of the feature model. Resource cost variability can be exploited to compare product performance under different cost models such as fixed cost, measured simulation cost, and worst-case cost.

The approach is demonstrated on an example using the MapReduce programming pattern as its common base product, and used to compare the performance of full versions to restricted demo versions of product variants. A restriction of our work is the concrete semantics which makes it difficult to reason about whole product lines, requiring a per-product approach to validation. This could be lifted by using a symbolic semantics and applying symbolic execution techniques to analyze the deployment sensitive SPL models, allowing the analysis to be lifted from concrete deployment architectures for specific products to a more generalized analysis.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: a cost and termination analyzer for ABS. In *PEPM'12*, pages 151–154. ACM, 2012.
3. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, and P. Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures. an industrial case study using Real-Time ABS. *J. of Service-Oriented Computing and Applications*, 2014. To appear.
4. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In *FM 2011*, *LNCS* 6664, pages 353–368. Springer, June 2011.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
6. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Inf.*, 50(2):77–122, 2013.
7. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
8. D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer, 2005.
9. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *SFM'11*, *LNCS* 6659, pages 417–457. Springer, 2011.
10. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *GPCE'10*, pages 13–22. ACM, 2010.
11. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In *FMCO'10*, *LNCS* 6957, pages 204–224. Springer, 2012.
12. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150. USENIX, 2004.
13. J. Garcia-Alonso, J. B. Olmeda, and J. M. Murillo. Architectural variability management in multi-layer web applications through feature models. In *FOSD'12*, pages 29–36. ACM, 2012.
14. H. Gomaa. *Designing Software Product Lines with UML.* Addison-Wesley, 2005.
15. A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. Delta-oriented architectural variability using Monticore. In *Software Architecture (ECSA'11), Companion Vol.*, page 6. ACM, 2011. Workshop on Software Architecture Variability (SAVA).
16. A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden. Hierarchical variability modeling for software architectures. In *SPLC*, pages 150–159. IEEE, 2011.
17. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
18. S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *ICSE'07*, pages 189–198. IEEE, 2007.

19. R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.
20. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'10*, *LNCS* 6957, pages 142–164. Springer, 2011.
21. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
22. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *ICFEM'10*, *LNCS* 6447, pages 646–661. Springer, Nov. 2010.
23. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *FoVeOOS'10*, *LNCS* 6528, pages 46–60. Springer, 2011.
24. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *ICFEM'12*, *LNCS* 7635, pages 71–86. Springer, Nov. 2012.
25. A. Kattepur, S. Sen, B. Baudry, A. Benveniste, and C. Jard. Variability modeling and QoS analysis of web services orchestrations. *ICWS*, pages 99–106. IEEE 2010.
26. R. Lämmel. Google's MapReduce programming model - revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.
27. N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Software Composition (SC'08)*, *LNCS* 4954, pages 36–51. Springer, 2008.
28. K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya. Using Coq in specification and program extraction of Hadoop MapReduce applications. In *SEFM'11*, *LNCS* 7041, pages 350–365. Springer, 2011.
29. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
30. I. Schaefer. Variability modelling for model-driven development of software product lines. In *VaMoS'10*, *ICB-Res. Rep.* 37, pages 85–92. Univ. Duisburg-Essen, 2010.
31. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC'10*, *LNCS* 6287, pages 77–91. Springer, 2010.
32. I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *AOSD'11*, pages 43–56. ACM, 2011.
33. I. Schaefer and F. Damiani. Pure delta-oriented programming. In *FOSD'10*, pages 49–56. ACM, 2010.
34. I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.
35. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *Software Tools for Technology Transfer (STTT)*, 14(5):477–495, 2012.
36. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Intl. Conf. on Requirements Engineering (RE'06)*, pages 136–145. IEEE, 2006.
37. R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.
38. D. Wagelaar and V. Jonckers. Explicit platform models for MDA. In *MoDELS'05*, pages 367–381, 2005.
39. F. Yang, W. Su, H. Zhu, and Q. Li. Formalizing MapReduce with CSP. *Intl. Conf. on Engineering of Computer-Based Systems*, pages 358–367. IEEE 2010.