Journal of Logical and Algebraic Methods in Programming 107 (2019) 108–135

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

On the modeling of optimal and automatized cloud application deployment

Stijn de Gouw [a], Jacopo Mauro [b,*], Gianluigi Zavattaro [c]

[a] Open University, the Netherlands
[b] University of Southern Denmark, Denmark
[c] FoCUS Research Team, University of Bologna/INRIA, Italy

A R T I C L E I N F O

Article history:
Received 9 October 2018
Received in revised form 7 June 2019
Accepted 7 June 2019
Available online 14 June 2019

Keywords:
Software system modeling
Declarative specification of deployment rules
Automatic cloud application deployment

A B S T R A C T

We investigate the problem of modeling the optimal and automatic deployment of cloud applications. We follow an approach based on three main pillars: (i) the specification of the computing resources needed by software components and those provided by the executing environment (e.g. virtual machines or containers), (ii) the declarative description of deployment rules, (iii) and the computation of an optimal deployment that minimizes the total cost by using constraint solving techniques. We experiment with such an approach by applying it to the *Abstract Behavioural Specification* language ABS, and we validate it by modeling and simulating with ABS (and its tool-suite) the Fredhopper Cloud Services, a worldwide system offering e-Commerce services, currently deployed on Amazon EC2.

1. Introduction

Software applications deployed and executed on cloud computing infrastructures should elastically adapt by dynamically acquiring or releasing computing resources. This is necessary to properly deliver to the final users the expected services at the expected level of quality, maintaining an optimized usage of the computing resources. For this reason, modern software systems call for novel engineering approaches that anticipate the possibility to reason about deployment already at the early stages of development. This is also testified by the appearance of software engineering approaches, like the DevOps approach to software development and delivery [1], that are strongly based on a closer collaboration among software developers, operations professionals, and quality assurance teams. In our opinion, an actual and successful integration among these different actors is not possible without having a common language, and common tools, for modeling and specifying software, for expressing deployment rules and constraints, and for describing service level agreements and the reaction to their violations.

Some projects, e.g., MODAClouds and DICE [2], already started the integration of aspects related to application deployment using high-level specification languages like UML. For instance, business requirements or expected quality of services can be expressed in such a way that they can be monitored at run-time. This allows one to detect deviations from the expected behavior and then trigger possible reactions. In this paper, we complement this line of research, by anticipating in the early stages of software development, techniques for the automatic and optimal deployment of cloud applications. More precisely, the idea is that of exploiting the expressed deployment desiderata and constraints in order to (i) automatically

* Corresponding author.
E-mail addresses: stijn.degouw@ou.nl (S. de Gouw), mauro@imada.sdu.dk (J. Mauro), gianluigi.zavattaro@unibo.it (G. Zavattaro).

https://doi.org/10.1016/j.jlamp.2019.06.001
2352-2208/© 2019 Elsevier Inc. All rights reserved.

synthesize deployment plans, that are guaranteed to be optimal w.r.t. some specific metrics, and (ii) integrate such deployment plans in the application specification, in such a way that formal reasoning is possible on a model of the deployed application. To this end, instead of considering UML, we adopt a language with a native formally defined and executable semantics that we can use as an example to simulate the automatically synthesized application deployment. Namely, we adopt the *Abstract Behavioural Specification* language ABS [3], which is an object-oriented specification language that, besides having an executable semantics, includes also a rich tool-chain supporting different kinds of static analysis (like, e.g., logic-based modular verification [4], deadlock detection [5], and cost analysis [6]).

In this paper we demonstrate that an actual integration between software specification/analysis and automatic/optimal deployment is possible, by presenting an ABS extension, that we call SmartDepl. This offers the possibility to enrich the specification of a software system in ABS with the information that usually drives the decisions taken by the operations professionals: the declarative expression of the logical deployment constraints and the computing resource requirements. This deployment information can be extracted from the specification, and given in input to a solver that computes the optimal allocation of software components to computing nodes, and synthesizes the corresponding deployment plans. These deployment plans are then integrated into the ABS specification in such a way that a model of the deployed application is obtained, on which the ABS simulator, as well as the various analysis tools (like, e.g., logic-based modular verification [4], deadlock detection [5], and cost analysis [6]) can be exploited for formal reasoning. It is worth noticing that the approach that we propose can be used not only for computing the initial application deployment, but also for synthesizing reconfiguration plans to be used to scale the specified application in case of, e.g., quality of service degradation.

ABS is an object-oriented specification language that supports the specification of asynchronously communicating concurrent objects, distributed over *deployment components* that provide objects with the computing resources they need to properly run. For our purposes, we adopted ABS because it allows for the modeling of computing resources and it has a real-time semantics reflecting the way in which objects consume resources. This makes ABS particularly suited for the modeling and for reasoning about cloud application deployment.

The approach followed by SmartDepl, to support the specification and analysis of issues related to deployment, is based on three main pillars:

- ABS classes can be enriched with annotations that indicate the computing resources (e.g., amount of memory or CPU cores) that are necessary in order to properly instantiate and execute objects of that class. Such annotations can also contain description of functional dependencies of objects of the annotated class, with respect to other objects that must be also deployed whenever a new object of that class is instantiated.
- A separate high-level constraint language, embedded in ABS, supporting the specification of declarative deployment rules, like the indication of the basic objects that must be present in a deployed system or the number of replica of a given service needed to guarantee a certain level of availability.
- An external solver that, based on the class annotations and the programmer's requirements extracted from the ABS specifications, generates ABS classes modeling the optimal *deployments*, i.e., classes that expose methods like `deploy` (and `undeploy`) that implement actual instantiation (and removal) of objects in new or already existing deployment components.

By adopting SmartDepl, it is actually possible to reason about deployment already during the early stages of software development. The novel language extensions allow the software designers to include deployment rules in their ABS specifications in such a way that all the analysis tools already present for ABS can be adopted to perform formal reasoning on the specified system. This is also supported by an easy-to-use integrated ABS development environment, called *collaboratory*,[1] available on the web (or easily downloadable and installable). In particular, the simulation facilities of ABS, available thanks to the executable semantics of the ABS specification language, can be used to simulate at the modeling level the impact of deployment rules and scaling policies. In this way, it is possible to avoid performing this kind of analysis by means of testing, on expensive and complex to manage run-time systems, and also to avoid time-consuming feedback loops, in case the testing phase points out the need to modify the already developed software. An additional advantage is that, by following the approaches of tools like ConfSolve [7] and Zephyrus [8], deployment is expressed declaratively and an external solver computes the actual detailed distribution of software components over computing resources. This approach has two advantages: the computed configuration is guaranteed to be correct (i.e., all the rules are satisfied) and optimal (i.e., there is no alternative configuration exploiting computing resources with a total smaller cost). In other terms, following our approach, the developed systems are correct- and optimal-by-construction (obviously, w.r.t. the deployment constraints and the costs inserted in the ABS specification).

Concerning the specific aspect of application scalability, it is interesting to observe that our approach goes beyond the current state of the art in scaling technologies (see e.g. [9]). For instance, container-based solutions like Kubernetes [10] exploit the so-called horizontal scaling approach: services can locally increase or decrease their number of instances depending on the values of monitored metrics (CPU average load, response time, ...). Other approaches (see, e.g., SmartScale [11] or the workload modeling engine proposed by Gandhi et al. [12]) complement horizontal with vertical scaling, i.e., the possibility

[1] http://abs-models.org/laboratory/.

to dynamically add/remove computing resources. These autoscaling techniques assume that the overall application architecture remains unchanged, and act on the number of instances of the services, or on the resources associated to the virtual machines used by the computing infrastructure. On the contrary, in our approach, we can consider re-deployment plans able to act on the application architecture. For instance, in case a peak of inbound requests is detected on the entry point of a pipeline of sequentially-interdependent services, our approach could evaluate the possibility to change the architecture: besides scaling out the entry point (so that more requests can be served in parallel) the subsequent services in the pipeline could be replaced with alternative implementations, designed on purpose to be highly available, as the workload of the entire pipeline is expected to increase. Moreover, by knowing all the modifications at the architectural level (i.e., all the service instances that are expected to be deployed during a re-deployment plan), it is possible to compute optimal deployment strategies where instances of different services share resources (e.g., they are installed in bulk on a new computing node added to the system). Such optimisation is impossible to achieve by solely relying on horizontal/vertical autoscaling [13].

Our work has been validated by modeling and analyzing the Fredhopper Cloud Services, an industrial case-study of the European FP7 Envisage project.[2] The Fredhopper Cloud Services offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). Depending on the specific profile of an e-Commerce company—like the expected number of clients or the preference between an 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 operations team which decides customized, hopefully optimal, service configurations taking into account several aspects like the level of replication of critical parts of the service to ensure high availability. The operators manually perform the operations to scale in or scale out the system and this usually causes the over-provision of resources for guaranteeing the proper management of requests during a usage peak.

We have used SmartDepl to realize an ABS specification of the Fredhopper Cloud Services that includes the knowledge and the current best practices of the operations experts: the deployment rules adopted to compute the initial deployment configuration, the actions usually taken to repair an unexpected virtual machine failure, as well as the scaling policies in case of service level agreement violations. This specification has been analyzed by means of the ABS simulation facilities. In particular, a visual representation of relevant metrics extracted from the simulation has been analyzed in strict collaboration with the operations expert at Fredhopper, who confirmed the faithfulness of the produced results. Also the initial system deployment, as well as the dynamic re-deployment procedures, computed by the automatic solver have been analyzed and validated by the operations team.

Structure of the paper. Section 2 discusses the related literature. Section 3 introduces the Fredhopper Cloud Services case-study. The description of the design and implementation of SmartDepl is in Section 4. The application of our technique to the Fredhopper Cloud Services use-case is reported in Section 5, while the analysis of the running times of the SmartDepl solver is reported in Section 6. Finally, in section 7 we draw some concluding remarks.

This paper finalizes the work started in [14,15] where the first version of SmartDepl has been presented. Compared with what presented in these papers, the current version of SmartDepl has been completely rewritten and extended to overcome the original limitations: new annotations are supported, for instance, to express preferences over bindings (e.g., useful to specify deployment optimization criteria based on geographical proximity), to indicate methods to be invoked on already present objects to notify the deployment of new objects (e.g., useful to break circularity in the object dependency relation), and to fix an ordering on these methods (e.g., useful to notify the deployment of the new object following a user-defined ordering). As a consequence of these modifications the parser of the ABS language has been updated. Another novelty is that the new parser now admits regular expressions in the denotation of deployment components (e.g., useful to express deployment properties on an entire class of components instead of a unique one). Also the code generation module of SmartDepl has been modified: the output now is directly ABS code (previously it was an ABS delta module). Finally, this paper presents a deeper validation of SmartDepl (e.g., including a completely new self-healing scenario of the considered case-study) and provides also an evaluation of its performance (completely absent in the previous papers).

2. Related work

With the increasing popularity of cloud computing, the problem of automating application deployment has recently attracted a lot of attention. Usually, the deployment task is conducted by a team of experts that establishes how the different components are to be installed and connected together. The deployment process is then automated by coding it in custom scripts. This approach is effective only if the architecture of the system is decided once and for all, and it is not expected to be customized for the different needs of the potential end-users, or shaped differently to, e.g., optimize the usage of the available computing resources.

Currently, developing an application for the cloud is accomplished by relying on the Infrastructure as a Service (IaaS) or the Platform as a Service (PaaS) levels. The IaaS offers a set of low-level resources forming a “bare” computing environment. Developers pack the whole software stack into virtual machines or containers containing the application and its dependencies and run them on physical machines of the provider’s cloud. Exploiting the IaaS directly allows a great flexibility

[2] http://www.envisage-project.eu/.

for the developer but requires also a great expertise and knowledge of the cloud and application entities involved in the process. At the PaaS level (e.g., [16,17]) a full development environment is provided. Applications are directly written in a programming language supported by the framework offered by the provider, and then automatically deployed to the cloud. The high-level of automation comes however at the price of flexibility: the choice of the programming language to use is restricted those supported by the PaaS provider, and the application code must conform to specific APIs. In this work we target the development of application at IaaS level, since it empowers the developers with greater control of the computing resources, thus allowing them to better optimize the application to be deployed.

Two deployment approaches standing at opposite sides are gaining more and more momentum: the *holistic* and the *DevOps* one. In the former, also known as *model-driven* approach, the software architect 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,[3] 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 major 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 [19], promoted by the OASIS consortium [20] for open standards. TOSCA proposes an XML or YAML like rich language to describe an application. Deployment plans are usually specified using the BPMN (Business Process Model and Notation) [21] or BPEL (Business Process Execution Language) [22] notations, workflow languages defined in the context of business process modeling.

The most important representative of the DevOps approach is Juju [23], by Canonical. It is based on the concept of *charm*: the atomic unit containing a description of a component. This description in form of meta-data is coupled with configuration data and *hooks* that are scripts to deploy and connect components. However, 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 does not specify the required functionality needed by a component. For instance, to install a WordPress blog in a basic scenario its only requirement is that the application should be connected to a database. However, Juju allows the deployment of the WordPress blog without warning that it should be deployed only after it has been properly connected to a database. This would actually result in a run-time error, occurring only after having “successfully” deployed WordPress.

In this paper we would like to anticipate the possibility to model and reason about deployment already at the early stages of development, thus proposing an approach that is in an intermediate level between the holistic and the DevOps one. In particular we would like to offer to the DevOps engineer all the flexibility to use and define ad-hoc components and a better and partially automated control on how to deploy them. On the one hand, compared to the DevOps approach, we allow for the (partial) automation of the deployment, bringing in evidence the connections between the different component and finding the optimal configuration to deploy. On the other hand, compared to the holistic approach, we do not require the specification of complex information related to the run-time behavior of the components that are required for the full automation of the deployment. The developer can indeed just focus on the relevant parts of the system in which deployment can be easily automatized, and then orchestrate and reason in one subsequent step about the deployment of the entire system.

Several approaches have been already proposed in the literature for modeling various aspects related to application deployment (see, e.g., the Bergmayr et al. survey [24] on the so-called CML, i.e. Cloud Modeling Languages). For instance, as far as the possibility to extend high-level specifications with business requirements and deployment desiderata, one of the main initiatives is supported by the MultiCloud Alliance [2] which is behind the ModaClouds and DICE projects. These are dedicated to the specification (in UML) and management of cloud application deployment abstracting away from the target cloud infrastructure. In this way cloud-portability and multi-cloud deployments are supported.

In the context of UML based software development, other ad-hoc languages or profiles have been developed to model specific aspects related with deployment. As an example, the MARTE profile [25] is dedicated to the specification of computing resources required by software components, while the OCL language [26], can be used to express constraints among UML elements. Differently from the above approaches, in this paper we mainly focus on the problem of automating application deployment with the aim of optimizing metrics like, e.g., the total cost of the computing nodes to be acquired. To achieve this goal, we need to use languages for describing resource usages and defining deployment constraints, by considering the right trade-off between their expressiveness and the possibility to effectively compute deployments that satisfy the constraints and optimize the metrics of interest. On the contrary, languages as those defined for UML usually are more focused on expressiveness. Nevertheless, we have encoded our constraint and resource description languages in a standard JSON format, that can be easily used as a target of translations from any other language like those mentioned above.

It is interesting to note that differently from the CMLs classified in the survey mentioned above [24], in this paper we propose a framework that combines: i) the possibility to express declaratively—at the specification level—the expected application deployments, ii) the automatic computation of corresponding deployment plans, and iii) the integration of such plans back in the initial specification. This allows the possibility to perform analysis like the simulation of the computed

[3] DevOps is a software development method that stresses communication, collaboration and integration between software developers and Information Technology professionals [18].

deployments. In our opinion, our contribution perfectly fits the last future direction mentioned at the end of the survey [24]: *"Simulation of deployment configurations. Analyzing and predicting non-functional properties such as costs and performance before the actual application provisioning is carried out"*.

In order to deal with automatic deployment the main source of inspiration for the development of SmartDepl was the Aeolus project [27,28], i.e., one of the first attempt to combine the holistic and bottom-up approaches studying the limits of what automation can achieve in the deployment process. As it turned out, in general the full automation of the deployment is undecidable and therefore there are limits on what can be automatically achieved. In this work, contrary to what has been done in the Aeolus project, we do not impose limitations on the components and their behavior trying to generate everything in a unique step, but we instead allow the user the flexibility to reuse and orchestrate different automatically generated deployment actions.

Other attempts to combine the holistic and bottom-up approaches are the management protocol approaches such as [29, 30] that establish protocols for reacting to failures by generating the sequence of actions to bring the configuration in a safe state. As for the Aeolus case, these works require the description of the behavior of the components through finite-state machines or more complex formalisms, thus imposing constraints on the components and their connections.

Many management tools for bottom-up deployment exist, e.g., CFEngine [31], Puppet [32], MCollective [33], and Chef [34]. Such tools allow for the declaration of components, by indicating how they should be installed on a given machine, together with their configuration files, but they are not able to automatically decide where components should be deployed and how to interconnect them for an optimal resource allocation, let alone the possibility to perform some reasoning on the deployment actions. These are usually the low-level tools used in the DevOps approach to deploy an application: our approach is at a higher level of abstraction.

Engage [35] is a deployment management framework consisting in (i) a language used to describe software component, computing resources and their dependencies, (ii) a configuration engine used to generate a full installation specification from an initial or partial configuration, and (iii) a deployment engine/run-time service used to carry out the installation and to manage the components during the deployment. Unfortunately, to ensure the existence of a feasible solution it introduces important simplifications: contrary to our approach, they rely on the acyclicity of the dependencies among components. This is crucial for Engage and precludes the possibility of having resources that are mutually dependent, as it can frequently happen in practice.

Another interesting project is ConfSolve [7] that consists basically of a definition of a *domain specific language* for describing configuration problems and a tool that uses constraint solving technology to solve them. ConfSolve is able to compute valid configurations that optimize one or more criteria like, e.g., maximizing the number of virtual machines per physical one. The ConfSolve language is object-oriented and declarative and allows using quantification and summation over decision variables in constraints. The major limitation of this approach is that the ConfSolve language models the problem of optimal provisioning (of virtual machines) rather than focusing on the deployment process. It does not take into account the wiring aspect, i.e., how to bind the components in use and which are the steps needed to reach the final (optimal) configuration computed by the solver. Other similar optimization approaches that address the problem of application placement on computing nodes are [36–39]. For a more extensive survey of the optimization approaches applied to software architectures we refer the reader to [40]. As pointed out in the survey, "the quality attributes, such as safety, maintainability, and security, have not been optimized with exact algorithms" and "due to the ever-increasing complexity of software systems and the growing number of design options, exact approaches usually are not suitable as optimization techniques; hence the lower number of papers employ these techniques". While in the survey exact methods approaches were applied only to the problem of service/component selection or to specific problems on embedded systems, in this work we chose to apply an exact method for the optimization of the definition and deployment of a cloud system, starting from a (partial) declarative specification of the goal.

VAMP (Virtual Applications Management Platform) [41,42] is a framework constituted by a language to describe the global structure of the application and an environment to manage the run-time deployment of components. The language extends the OVF (Open Virtualization Format) [43] language, a proposed standard for a uniform format for applications to be run on virtual machines. The VAMP deployment process is implemented as a decentralized protocol in a self-configuration manner. The approach is interesting but limited for our purposes as it works under the assumption that the dependency graph is acyclic and requires the developer to specify the virtual machine in which a given component lives. Following similar philosophy and limitations, we can mention Terraform [44], JCloudScale [45], Apache Brooklyn [46], and tools supporting the Cloud Application Management for Platforms protocol [47] and deployable on IaaS or clustering solutions such as [48,49].

SmartFrog [50] is a Java framework, developed at HP, for managing deployment in a distributed setting. It shares some similarities with the Engage approach as every component has a declarative description. It lacks, however, a way to use the declarative description to extract some information for the deployment plan or to perform some static checks. DADL (Distributed Application Description Language) [51] is a language extension of SmartFrog that enables to express different kinds of constraints (such as Service Level Agreements SLAs and elasticity). This work, however, just focuses on the language aspects.

CloudFoundry [52] is a PaaS solution by VMware that, taking advantage of the latest innovations from container technologies such as Docker [53] and Kubernetes [49], allows the selection, connection and pushing to a cloud of well defined services (databases, message buses, etc.), used as building blocks for writing applications with one of the supported infras-

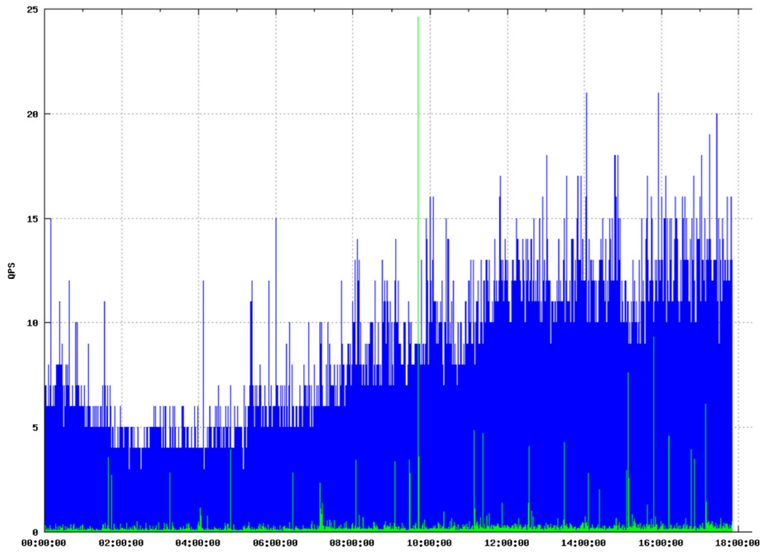


Fig. 1. Number of queries per second (in green the query processing time). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

tructures. Contrary to our approach, the platform allows only a predefined set of service to be used directly, with a limited set of operations that can be executed upon them. Moreover, it does not allow any reasoning or optimization about the final configuration.

3. The Fredhopper Cloud Services

In this section we give an overview of the case-study used to validate our approach.

The company Fredhopper provided the Fredhopper Cloud Services[4] to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). At the time of the case study, Fredhopper Cloud Services powered over 350 global retailers with more than 16 billion in online sales every year.

The services offered by Fredhopper are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Software services are deployed as *service instances*. Each instance offers the same service and is exposed via Load Balancer endpoints that distribute requests over the service instances.

The number of requests can vary greatly over time, and typically depends on several factors. For instance, the time of the day in the time zone where most of the end-users are plays an important role (typical lows in demand are observed between 2 am and 5 am). Fig. 1 shows a real-world graph for part of a single day plotting the number of queries per second over the time of the day. The data samples were collected starting at midnight up until 18:00 o'clock. The figure shows that the number of requests are the lowest between 2:00 am–5:00 am.

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

Fig. 2 shows a block diagram of the Fredhopper Cloud Services. We briefly explain the architecture.

Load Balancing Service The Load Balancing Service is responsible for distributing requests from the service endpoints to their corresponding instances. Currently at Fredhopper, this service is implemented by HAProxy,[5] a TCP/HTTP load balancer.

Service Instance Service instances provide a specific service to customers. Examples of services are a query service that enables users to request or search for information (i.e., if an end user searches for a term, the query service yields a list of matching products) and a data service to allow customers to update and configure their product catalog.

[4] Fredhopper was recently acquired and integrated into the ATTRAQT Group plc, see http://www.fredhopper.com.
[5] www.haproxy.org.

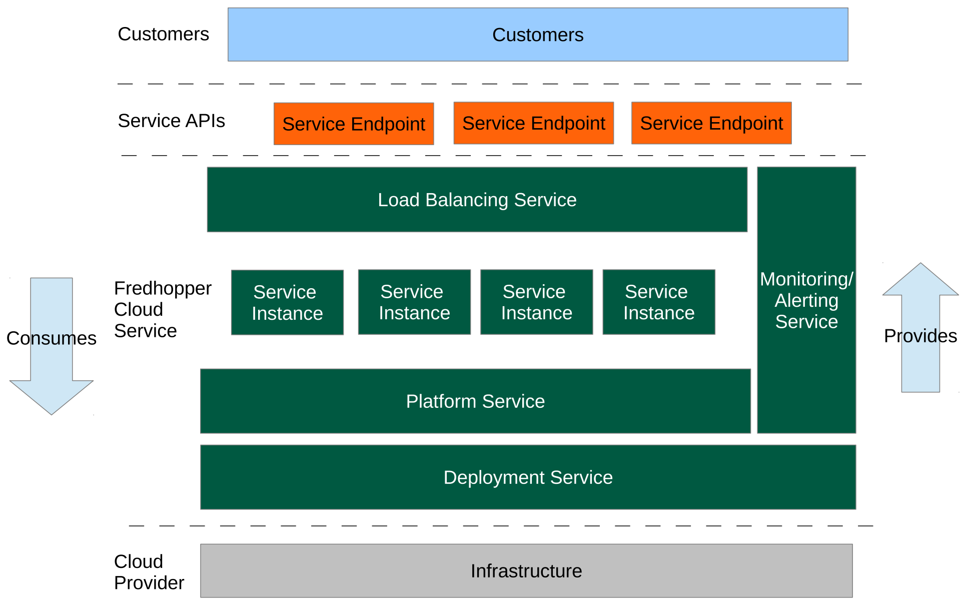


Fig. 2. The architecture of the Fredhopper Cloud Services.

Platform Service The Platform Service provides an interface to the Cloud Engineers to manage customer information, deploy and manage service instances associated to customers, and associate service instance to endpoints (load balancers).

Deployment Service The Deployment Service provides an API to the Platform Service to deploy service instances (using a dedicated Deployment Agent) onto specified virtualized 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 virtualized resources. At the time of writing, the Fredhopper Cloud Services utilizes virtualized resources from the Amazon Web Services,[6] where processing and memory resources are exposed through Elastic Compute Cloud instances.[7]

Monitoring and Alerting Service The Monitoring and Alerting Service provides 24/7 monitoring services on the functional and non-functional properties of the services offered by the Fredhopper Cloud Services, the service instances deployed by the Platform Service, and the healthiness of the acquired virtualized resources. If a monitored property is violated, an alert is raised to the Cloud Engineers via email and SMS messages, and Cloud Engineers can react accordingly.

The deployment of the Fredhopper Cloud Services follows requirements originated from both business decisions or technical reasons. For instance, for security reasons, services that operate on sensitive customer data should not be deployed on machines shared by multiple customers. We now list some specific relevant deployment requirements.

- To increase fault-tolerance, we aim to spread virtual machines across geographical locations. Amazon allows specifying the desired region (a geographical area) and availability zone (a geographical location in a region) for a virtual machine. Fault tolerance is then increased by balancing the number of machines between different availability zones. Thus, when scaling, as application requirement, the number of machines should be adjusted in all zones simultaneously. Effectively this means that in a region with two availability zones, we scale (increase or decrease) with an even number of machines.
- Each instance of a Query service is in one of two modes: ‘live’ mode to serve queries, or a so-called ‘staging’ mode. Updates to the product catalog and generating the indexes on the catalog for faster searching can only be done by an instance in staging mode. There always should be at least one instance of Query service in staging mode.
- The network throughput and latency between the Platform Service and indexer is important. Since the infrastructure provider gives better performance for traffic between instances in the same zone, we require the indexer and the Platform Service to be in the same zone.

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

- Installing an instance of the Query Service requires the presence of an instance of the Deployment Service on the same virtual machine.
- The Load Balancer layer is a separated tier with dedicated computing nodes (for performance reasons and fault tolerance), that is: load balancer endpoints require a dedicated machine without other services co-located on the same virtual machine.

In addition to the above *logical* deployment requirements, the virtual machine(s) on which service instances are deployed and executed should satisfy certain *resource requirements*. The exact requirements typically depend on the kind of service (i.e., a Load Balancer endpoint requires a VM with high network throughput), the configuration of the service instance, and customer-dependent data such as usage patterns and the size of their product catalog. Resource requirements include:

- The Query Service is single-tenant service (i.e., its instances are dedicated to a single customer).
- With normal usage, historical data from log files of the in-production system show that the Query Service requires a virtual machine with at least two-core CPU and 3 GB of memory.
- Under heavy workloads (i.e., during promotions), the Query Service requires a virtual machine with a four-core CPU and 4500 MB of memory.

Finally, to realize a target deployment configuration, certain installation actions should be carried out in a specific order, and (counterpart) removal actions are needed to remove service instances. The list below shows the most important installation actions (in order) that should be carried out to deploy a new Query Service instance.

1. The new Query Service instance must notify its existence to the Platform Service. The Platform Service then assigns it a unique service ID which is used in other services.
2. The Query Service instance must notify a Deployment Service instance running on the same VM, so that the query service instance can be properly configured and managed (i.e., started, stopped, etc.).
3. The Query Service instance must notify all load balancer endpoints *in the same region* that the instance can be used (i.e., the balancers may start forwarding requests to it).

4. SmartDepl: design and implementation

In this section we present SmartDepl, our extension of the *Abstract Behavioural Specification* language ABS for the modeling of cloud application deployment. As anticipated in the Introduction section, SmartDepl is based on (i) annotations used to describe functional dependencies and the resources consumed by objects, as well as the resources provided by the deployment components hosting and executing such objects, a (ii) declarative language used for expressing global deployment requirements, and (iii) an external solver that computes optimal distributions of objects on deployment components, and generates the ABS code necessary to reach such configurations. We first briefly recall the main characteristics of ABS, and then we detail the above three aspects in three separate subsections.

4.1. Introduction to ABS

ABS is an object-oriented specification language with a formally defined and executable semantics, that can be used to simulate the specified system already at the early stage of software development. It also includes a rich tool-chain supporting different kinds of static analysis (like, e.g., logic-based modular verification [4], deadlock detection [5], and cost analysis [6]). Executable code can be automatically obtained from ABS specifications by means of code generation. The ABS language supports the specification of asynchronously communicating concurrent objects, distributed over *deployment components* corresponding to containers offering to objects the resources they need to properly run. Below, we will recap the specific linguistic features of ABS to support the modeling of the deployment and the cost annotations required by our approach. ABS has a semantics for the cost annotations. Simulation and code generation tools take this semantics into account during execution of an ABS model. This makes it possible to simulate the resource usage of the program being modeled. For further details on ABS, we refer the interested reader to the ABS project website [3].

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

See, for instance, the following code excerpt:

```
CloudProvider cp = new CloudProvider("Amazon");
cp.addInstanceDescription(Pair("c3.xlarge",
   map[Pair(CostPerInterval,210), Pair(Cores,4),
        Pair(Memory,750), Pair(PaymentInterval,3600000)]));
DeploymentComponent dc = cp.prelaunchInstanceNamed("c3.xlarge");
[DC: dc] Service s = new QueryServiceImpl( ... );
```

In the ABS code above, the cloud provider “Amazon” is modeled as the object `cp` of type `CloudProvider`. The fact that “Amazon” can provide a virtual machine of type “c3.xlarge” is modeled by calling `addInstanceDescription` in Line 2. With this instruction we also specify that c3.xlarge virtual machines

- have a cost of 0.210 per hour (using time units of 10 ms, a cost of 210 cents per interval, and intervals of 360000 time units)
- provide 7.5 GB of RAM
- offer 4 cores.

In Line 5 an instance of “c3.xlarge” is launched and the corresponding deployment component is saved in the variable `dc`. Finally, in Line 6 , a new object of type `QueryServiceImpl` (implementing interface `Service`) is created and deployed on the deployment component `dc`.

ABS supports declaring interface hierarchies and defining classes implementing them.

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

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

4.2. ABS annotations

SmartDepl relies on annotation of ABS classes to state their costs and their requirements. To facilitate the inter-operability between ABS and possible external tools (like, e.g., possible future graphical environments for deployment issues specifications) we have adopted a JSON syntax for the cost annotations.[8] In particular, the definition of the cost annotation is provided in a JSON string as follows:

```
[SmartDeployCost : JString ]
```

The JSON string `JString` defines the costs. An example that describes two possible deployment scenarios for objects of class `QueryServiceImpl` is the following.

```
{ "class" : "QueryServiceImpl",
  "scenarios" : [
  { "name" : "staging",
    "provide":-1,
    "cost": { "Cores": 2, "Memory" : 700},
    "sig": [ { "kind" : "require",
               "type" : "DeploymentAgent"},
             { "kind" : "constant",
               "value" : "True"}],
    "methods" : []},
  { "name" : "live",
    "provide":-1,
    "cost": { "Cores": 1, "Memory" : 300 },
    "sig": [ { "kind" : "require",
               "type" : "DeploymentAgent"},
             { "kind" : "constant",
               "value" : "False"}],
    "methods" : []}
  ]}
```

Listing 1: Example of deploying scenario annotation.

The first part of the annotation models the deployment of a Query Service in staging mode, the second one models the deployment in live mode. A Query Service in staging mode requires 2 cores and 7 GB of RAM. This is needed for the additional functionality that the staging mode offers (such as constructing indexes for the product catalog). In live mode, 1 core and 3 GB of RAM suffices. Creating a Query Service object requires the instantiation of its two initialization parameters: a Deployment Agent object and a Boolean value representing the staging modality. The first parameter is required: this means that the Query Service requires a reference to an object of type `DeploymentAgent` passed via the first initialization parameter. The second parameter should instead be instantiated with `True` or `False` depending on the deployment scenario.

[8] The JSON schema of the annotation is available at https://github.com/jacopoMauro/abs_deployer/blob/master/spec/smart_deploy_cost_a nnotation_schema.json.

We require an annotation for every relevant class that can be involved in the automatic generation of the initial configuration or the scaling up procedures. Intuitively, an annotation for the class `C` describes: (i) the maximal resource consumption of an object `obj` of 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), (iii) how many other objects in the deployed system can use the functionality provided by `obj`, and (iv) additional references that may be added by invoking a class method.

In particular, for every annotation, the keyword `class` is used to define the class `C` for which we want to define the costs, while the keyword `scenarios` contains a list of the possible deployment scenarios for an object of that class. Every scenario specifies the following information:

- `"name": X`. Associates a name `X` to the deployment scenario.
- `"provide": 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 functionality only to a limited number of other client objects. If an unlimited number of client objects can be provided for, the provide value can be set to -1 (as in the case of the annotation for the `QueryServiceImpl` class).
- `"cost"`. Indicates the resource cost of an object `obj` of class `C`. The resources available are those defined in the ABS Cloud API. In the `QueryServiceImpl` annotation two only resources are used: `Cores` representing the number of required processors, and `Memory` representing the required amount of RAM.
- `"sig"`. Indicates how the initialization parameters for class `C` must be instantiated when an object `obj` of class `C` is deployed. There are three different cases:
  1. `"kind": "constant"` indicates that the parameter must be set to the default value specified by using the keyword `"value"`.
  2. `"kind": "require"` indicates that the parameter is required to be instantiated by SmartDepl during the deployment code generation phase. Here, SmartDepl is responsible to first create an object having the interface specified with keyword `"type"` and then pass it as a parameter when `obj` is instantiated.
  3. `"kind": "list"`: the parameter requires a list of at least a given number of elements (defined by using the keyword `"num"`). Similarly to what happens in the `"require"` case, these objects need to have the interface specified with keyword `"type"` and should be defined by SmartDepl.
- `"methods"`. Indicates the possibility to add or remove references to objects by invoking methods of the class `C`. This is needed to capture faithfully situations where interdependent objects need to be deployed. For instance, in the Fredhopper Cloud Services a Deployment Agent objects needs a reference to a Query Service object. However, at the same time, the Query Service object needs a reference to a Deployment Agent object. If the references are passed only at object creation, since objects can not be deployed simultaneously, it would be impossible to deploy these two objects. Adding the possibility to add references even after their creation breaks instead the circularity of dependencies, thus allowing for the creation of these two objects. The following is an example of the use of the keyword `"methods"`.

```
{ "class" : "DeploymentAgentImpl",
  "scenarios" : [
  { "name" : "default",
    "provide":-1,
    "cost":  { "Cores": 1, "Memory" : 80 },
    "sig": [],
    "methods" : [
      { "add" :
        { "name": "installDA",
          "param_type": "Service" },
          "remove" : {"name": "uninstallDA"}}]
  }]}
```

Here, SmartDepl is informed that in order to create an object as instance of the class `DeploymentAgentImpl` no parameter is required. When the object is created, however, it is possible to invoke its `installDA` method to add a reference to an object with interface `Service` (e.g., a Query Service). This therefore allows the creation of a `DeploymentAgentImpl` object first. Then it is possible to pass to it the reference to a deployed Query Service by invoking the `installDA` method. When the Query Service needs to be removed, its reference can be canceled from the `DeploymentAgentImpl` object by invoking the `uninstallDA` method.

In ABS, the *deployment components* (DCs) are the units responsible for offering to the objects the computing resources they require. As for the class cost annotations, the definition of the DC types is provided in a JSON string as follows.

```
[SmartDeployCloudProvider: JString ]
```

The JSON string `JString` defines the JSON objects listing all the DC types and their properties.[9]

[9] The JSON schema of the annotation is available at https://github.com/jacopoMauro/abs_deployer/blob/master/spec/smart_deploy_cloud_ provider_annotation_schema.json.

Table 1
JSON example to specify three DC types.

```
1 { "c4_xlarge_us": {
2     "cost":209, "payment_interval": 1,
3     "resources": { "Cores":4, "Memory":750 }},
4   "c4_xlarge_eu": {
5     "cost":209, "payment_interval": 1,
6     "resources": { "Cores":4, "Memory":750 }},
7   "c4_2xlarge_eu": {
8     "cost":419, "payment_interval": 1,
9     "resources": { "Cores":8, "Memory":1500 }}}
```

Table 2
DRL grammar.

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

As an example, Table 1 presents the JSON definition of three types of virtual machine corresponding to the “xlarge” and “2xlarge” instances of the Compute Optimized instances (version 4) of Amazon EC2 Instance Types.[10]

Every DC type has a name. In Line 1 for instance we define a DC type called `"c4_xlarge_us"` that represents an instance `xlarge` of type Compute, version 4 deployed in the US region. The name identifies the DC type and, as shown later, it can be used to distinguish the different instances (e.g., to filter the instances deployed in Europe from those deployed in US). Every instance has associated a cost and a payment interval. At Line 2, for instance, the instance `"c4_xlarge_us"` has been associated a cost of 209 for every payment interval corresponding to 1 ABS logical time unit. The last ingredient to finalize the description of the DC type is the amount of resources that it can provide. These are defined by using the keyword `"resources"`. At Line 3, for instance, we define that an instance of type `"c4_xlarge_us"` provides 4 cores and 7.5 GB of memory. The resources that can be used are those defined by the ABS Cloud API.[11]

Similarly to what done in Lines 1-3, in Lines 4-6 and 7-9 we define two other instance types. The first is a `xlarge` similar to the previous one but in this case with a different name, meaning that it is deployed in Europe instead of the US region. The last DC type instead is a completely different kind of DC having the double amount of cores and memory, but at a higher cost.

4.3. Declarative deployment specification

Computing a deployment configuration requires taking into account the expectations of the ABS programmer. For example, in the Fredhopper Cloud Services, one initial goal is to deploy with reasonable cost a given number of Query Services, possibly located on different machines to improve fault tolerance, and later on to scale the system according to the monitored traffic. Each requirement can be expressed in SmartDepl by considering two types of deployment requirements: constraints about the distribution of objects over deployment components, and preferences concerning relationships among the objects. In the first case, the expressed constraints must be satisfied by the declaratively specified deployment, while in the second case the expressed preferences indicate additional conditions that should be satisfied if it is possible.

The language for expressing the strictly required constraints is the *Declarative Requirement Language* DRL.

As shown in Table 2, that reports an excerpt of the DRL grammar,[12] a requirement is a (possibly quantified) Boolean formula `b_expr` obtained by using the usual logical connectives over comparisons between arithmetic expressions. An atomic arithmetic expression is an integer (Line 6), a sum statement (Line 8) or an identifier for the number of deployed objects

[10] https://aws.amazon.com/ec2/instance-types/.
[11] For more information, see the tutorial available at http://abs-models.org/smartdepl-tutorial/.
[12] The complete grammar defined using the ANTLR compiler generator is available at https://github.com/jacopoMauro/abs_deployer/blob/smart_deployer/decl_spec_lang/D eclSpecLanguage.g4.

(Line 9). The number of objects to deploy using a given scenario is defined by its class name and the scenario name enclosed in square brackets (Line 12). For example, the below formula requires deploying at least one `QueryServiceImpl` object in staging mode.

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

The square brackets are optional (Line 12 - first option) for objects with only one default deployment scenario. Regular expressions (`RE` in Line 12) can match objects deployed using different scenarios. The number of deployed objects can be prefixed by a deployment component identifier to denote just the number of objects defined within that specific deployment component. As an example, the deployment of only one object of class `DeploymentServiceImpl` on the first and second instance of a "c4" virtual machine can be enforced as follows.

```
c4[0].DeploymentServiceImpl = 1 and
  c4[1].DeploymentServiceImpl = 1
```

Here the 0 and 1 numbers between the square brackets represent respectively the first and second virtual machine of type "c4". To shorten the notation, the `[0]` can be omitted (Line 9).[13]

It is possible to use also quantifiers and sum expressions to capture more concisely some of the desired properties. Variables are identifiers prefixed with a question mark. As specified in Line 13, variables in quantifiers and sums can range over all the objects (`'obj'`), all the deployment components (`'DC'`), or just all the virtual machines matching a given regular expression (`RE`). In this way it is possible to express more elaborate constraints such as the co-location or distribution of objects, or to limit the amount of objects deployed on a given DC. As an example, the constraint enforcing that every Query Service has a Deployment Service installed on its virtual machine is as follows.

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

Listing 2: Co-location example requirement.

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

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

Listing 3: Requirement that load balancers should run on a dedicated VM.

The DRL is used to specify the constraints over the objects to be created; we now present a complementary language used to describe additional preferences concerning the way the deployed objects should be interconnected. This is extremely useful to connect, for instance, all the load balancers deployed in a region with only the back-end services deployed on the same region, or to require that a Query Service must use the Deployment Agent deployed on the same deployment component.

SmartDepl allows the passing of an object reference o to an object o' in two different ways:

- by passing o as an instantiation parameter (when invoking the new method that creates the object o'),
- by invoking `o'.m(o)` if the cost annotation of the class of o' allows the call of the method `m` to add the reference of o.

In both these cases we say that o is used by o'.

The previous constraints are used to compute possible configurations. Among all the configurations that satisfy the constraints, if any, the SmartDepl solver produces the one that minimizes the cost of the DC used. In case of ties, the SmartDepl solver minimizes the number of created objects. Once this optimal configuration is obtained, the user can use SmartDepl to state preferences on how the references among objects should be distributed. The grammar to express a preference is defined in Table 3.

A preference may be either the string `local` or an arithmetic expression (Line 1). The `local` preference is used to maximize the number of references shared among the components deployed in the same deployment component. Stating this preference, for instance, will enforce that if a Query Service and a Deployment Agent are deployed on the same deployment component, then the Query Service will use that Deployment Agent, and vice versa.

[13] We assume that every deployment requirement expressed in DRL deals with only a bounded number of deployment components (the bound is a configuration parameter for SmartDepl). Notice that this does not mean that the total number of deployment components in an application is fixed as, for instance, a scale-in or scale-out deployment action can be repeated an unbounded number of times.

Table 3
Grammar to express preferences over shared object references (missing non terminals are as defined in Table 2).

```
 1 preference: 'local' | expr ;
 2 term : INT                                |
 3     VARIABLE 'used' 'by' VARIABLE         |
 4    ('exists' | 'forall') VARIABLE ('of' 'type' objId)?
 5      'in' typeV ':' b_expr                |
 6    'sum' VARIABLE ('of' 'type' objId)?
 7      'in' typeV ':' expr                  |
 8    '(' b_expr ')'                         ;
 9 objId :  ID | ID '[' ID ']' | ID '[' RE ']' ;
10 typeV : 'DC' | RE ;
```

Arithmetic expressions are used instead to capture more advanced preferences. These expressions are built by using as basic atoms integers (Line 2) and the predicate `?x used by ?y`, which is assumed to be evaluated to 1 if the object referenced by the variable x is used by the object referenced by the variable y, 0 otherwise. In order to instantiate the variables of the predicate `used by`, quantifiers (Lines 4-5) and sum expressions (Line 6-7) may be used. As an example, the following query is used to maximize the number of Query Services deployed in Europe in staging mode that are connected to all the Load Balancers deployed in Europe.

```
sum ?x of type
 QueryServiceImpl['staging'] in '.*_eu' :
 forall ?y of type
  LoadBalancerEndPointImpl in '.*_eu' :
  ?x used by ?y
```

In the first two lines we use the `sum` expression to match to the variable `?x` all the `QueryServiceImpl` deployed in staging mode hosted by a deployment component whose name matches the regular expression `'.*_eu'`. Similarly, in the third and fourth lines we use the `forall` expression to match to the variable `?y` all the `LoadBalancerEndPointImpl` deployed in Europe. The `forall` expression is evaluated to 1 if, fixing the possible assignments of the variable `?y`, the predicate `?x used by ?y` is true. If instead there is an instance of `LoadBalancerEndPointImpl` that is not used by the object `?x` than the forall expression returns 0. Due to the fact that the first expression is a sum expression, the final behaviors of the preference is to maximize the number of instances of `QueryServiceImpl` deployed in Europe in staging mode that are used concurrently by all the `LoadBalancerEndPointImpl` objects deployed in Europe.

With a forall, exists and sum expression, objects can be filtered by their name and scenario. Regular expressions can be used to match the scenario name. For instance, in the first line, we could have required to match to the variable `?x` all the Query Services simply replacing the `'staging'` regular expression with `'.*'`. The identifiers `ID` here could be the name of the class of the objects to match.

For example, assuming that we have already deployed an object called `obj` (in our case-study being an instance of type `LoadBalancerEndPoint`) we can maximize all the new Query Services created in Europe that use `obj` as follows.

```
sum ?x of type QueryServiceImpl['.*'] in '.*_eu' :
 exists ?y of type obj in DC : ?x used by ?y
```

Here, instead of specifying after the keywords `of type` a class name and a scenario we specify directly the name of the object. We used the keyword `DC` to indicate that we want to match `obj` to the variable `?y` wherever this object is deployed. `DC` (Line 15) stands indeed for the set of all the possible deployment components.

4.4. Deployment engine

In SmartDepl automatic deployment is realized through a *deployment engine* that receives in input the deployment annotations added to an ABS specification, and produces in output ABS code that models an optimal deployment satisfying the constraints at a minimal total cost. In the following, for simplicity, we use the term SmartDepl to define both the formal extension of ABS but also the solver that runs to produce the ABS desired code.

The key idea of SmartDepl is to allow the user, on the one hand, to declaratively specify the desired deployments and, on the other hand, to develop its program abstracting from concrete deployment decisions. More concretely, specific types of deployment are specified as program annotations. These annotations are processed, and for each of them the deployment engine generates a new ABS class that exposes methods specifying the low-level deployment steps needed to reach (or to undo) the desired target deployment. Then this class can be used to trigger the execution of the deployment, and to undo it in case the system needs to scale in (i.e., terminate the deployed instances).

As an example, imagine that an initial deployment of the Fredhopper Cloud Services has been already obtained and that, based on a monitor decision, the user wants to add a Query Service instance in live mode. The annotation that describes this requirement is the JSON object defined in Listing 4.[14]

```
 1 { "id": "AddQueryDeployer",
 2   "specification": "QueryServiceImpl[live] = 1",
 3   "obj": [
 4     { "name":"platformServiceObj",
 5       "interface":"MonitorPlatformService",
 6       "provides":[ {
 7         "ports":[ "MonitorPlatformService",
 8                   "PlatformService" ],
 9         "num":-1 } ],
10       "methods" : [ {
11           "add" : {
12             "name": "addEndPoint",
13             "param_type": "LoadBalancerEndPoint" },
14           "remove" : {
15             "name": "removeEndPoint",
16             "param_type": "LoadBalancerEndPoint" }}
17       ... ],
18   "add_method_priorities":[
19     { "class":"loadBalancerEndPointObj",
20       "method":"add" },
21     { "class":"platformServiceObj",
22       "method":"addServiceInstance" },
23     ... ],
24   "bind preferences":[
25     "local",
26     "sum ?x of type QueryServiceImpl['.*'] in 'DC' :
27       exists ?y of type platformServiceObj in DC :
28         ?x used by ?y",
29     ... ],
30   "DC": [] }
```

Listing 4: An example of a deployment annotation.

In Line 1, the keyword `"id"` specifies that the name of the class with the deployment code, to be synthesized by SmartDepl, is `AddQueryDeployer`. As we will see later, this class exposes methods to be invoked to actually execute deployment actions that modify the current deployment according to the requirements in the deployment annotation. The second line contains the declarative specification of the desired configuration in DRL. Deploying a new instance of the Query Service may involve other relevant objects from the surrounding environment, such as the `PlatformService`. Which objects are relevant may come from business, security or performance reasons, thus in general it may be undesirable to select or create automatically a Service instance of the right type. SmartDepl is flexible in this regard: the user supplies the appropriate ones. By using the keyword `"obj"`, Lines 3-17 list the appropriate objects. Since these objects are already available, they need not be deployed again. As an example, in Listing 4 a Platform Service object is assumed to be already deployed. The name of the object is specified with the keyword `"name"` (Line 4). As detailed later, this name is the formal parameter used to identify the existing object within the class generated by SmartDepl. It just needs to be a fresh name, i.e., a string not used as a class or interface name within the main ABS program. The interface implemented by the object is defined by using the keyword `interface` (Line 5). All the interfaces provided by the object that could be used by other objects are then specified by using the keyword `"ports"` (Line 6). In this case the object provides two interfaces: the interface `MonitorPlatformService` it implements, as well as the `PlatformService` interface (i.e., an interface that is extended by `MonitorPlatformService`). The amount of other objects that can use these interfaces is defined by the keyword `"num"` (Line 9)—in this case a -1 value means that the object can be used by an unbounded number of other objects. With the keyword `"methods"` it is possible to specify how additional references may be added to the existing object. In this case it is specified that the method `addEndPoint` and `removeEndPoint` (Line 12-15) can be used to add and remove the reference to an object implementing the interface `LoadBalancerEndPoint`. If the methods for adding references need to be invoked following a given order, priorities can be assigned to such methods by using the keyword `"add_method_priorities"` (Line 18). The methods are given in a list, higher priority first. Every method is defined by its name (keyword `method` in Lines 20 and 22) and the class or the already available object implementing it (keyword `class` in Line 19 and 21). In Listing 4 we state that the method `add` of the existing object `loadBalancerEndPointObj` must be invoked before the method `addServiceInstance` of the object `platformServiceObj`. In case also the methods for removing references need to be executed in a given order, the priority of the removal methods can be encoded in a similar way by using the keyword `"remove_method_priorities"`.

[14] As done for previous annotations, we have adopted a JSON syntax for the deployment annotations. The JSON schema is defined in https://github.com/jacopoMauro/abs_deployer/blob/master/spec/smart_deploy_annota tion_schema.json.

The preferences over how references are shared among objects are defined using the keyword `bind preferences`. Preferences are given in a list, higher importance first. The grammar used by expressing the preference is exactly the one defined in Section 3.

Finally, with the keyword `"DC"` (Line 30), the user specifies if there are existing deployment components with free resources that can be used to deploy new objects. In this case, e.g., for fault tolerance reasons, the user wants to deploy the Query Service in a new machine and therefore the `"DC"` is empty. Otherwise, the list of the existing DC names with their available resources have to be provided.

Once the annotation is given, the user may freely use the corresponding class in the main ABS code. For instance, the below ABS code scales the system in or out based on a monitor decision.

```
1 while ( ... ) {
2  AddQueryDeployer depObj = new AddQueryDeployer(
3       cProv, platformService, loadBalancerService, serviceProvider);
4  if ( monitor.scaleUp() ) {
5     depObj.deploy();
6   } else if (monitor.scaleDown()) {
7      depObj.undeploy(); } }
```

In the first line a new object is instantiated of the class `AddQueryDeployer` (this class is generated automatically by SmartDepl). The first parameter is the cloud provider, as defined for instance in Section 4.2. The next parameters are the objects already available for the deployment that do not need to be re-deployed. These are given according to the order they are defined in the annotation `obj` in Listing 4. The generated class implements: i) a `deploy` method to realize the deployment of the desired configuration, ii) an `undeploy` method to undo the deployment action gracefully by removing the virtual machine created by the last `deploy` method, iii) getter methods to retrieve the list of new objects and deployment components created by the `deploy` method (e.g., a call `depObj.getIQueryService()` retrieves the list of all the Query Services created by `depObj.deploy()`). The actual addition of the Query Service is performed in Line 5 with the call of the `deploy` method. The DCs and the objects created in every call of the deploy method are saved in a stack. If the monitor decides to scale in (Line 6), the last deployment solution is undeployed (Line 7) by calling the `undeploy` method.[15]

As an example, the excerpt of the code generated by SmartDepl for the annotation defined in Listing 4 is the following. Please note that, for presentation purposes, the fresh names generated by SmartDepl to identify objects and DCs have been renamed.[16]

```
 1 DeploymentComponent dc1 =
 2    cloudProvider.prelaunchInstanceNamed("m4_large_eu");
 3 ls_DeploymentComponent =
 4    Cons(dc1,ls_DeploymentComponent);
 5 [DC: dc1] DeploymentAgent o1 = new DeploymentAgentImpl();
 6 ls_DeploymentAgent = Cons(Pair(o1,dc1),ls_DeploymentAgent);
 7 [DC: dc1] IQueryService o2 = new
 8    QueryServiceImpl(o1,False);
 9 ls_IQueryService = Cons(Pair(o2,dc1), ls_IQueryService);
10 ls_Service = Cons(Pair(o2,dc1),ls_Service);
11 ls_EndPoint = Cons(Pair(o2,dc1),ls_EndPoint);
12 loadBalancerEndPointObj.addLBE(o2);
13 platformServiceObj.addServiceInstance(o2);
14 o1.installDA(o2);
15 deploymentServiceObj.addDS(o1);
```

Listing 5: Example of generated code.

At Line 2, a new deployment component `dc1` is created by using the functionality `prelaunchInstanceNamed` of the ABS Cloud API. This models the creation of a virtual machine corresponding to a Compute Instance `large` of Amazon. Among all the possible choices of DC types, SmartDepl has selected this one because it is the cheapest one. Since we did not specify in the `specification` string where these machines need to be deployed, SmartDepl selected randomly the European region instead of the US one.

At Line 3 the bookkeeping variable `ls_DeploymentComponent` is updated with the list of the newly created DC.

In Line 5 an object of class `DeploymentAgent` is created, since every Query Service requires a corresponding `DeploymentAgent` (it is one of the required parameters, cf. Section 4.2) to be deployed before the Query Service. In Lines 7-8 the desired object of class `IQueryService` is created. Both objects are deployed on `dc1`. As happened at Line 3 for the DC, Lines 6 and 9-11 update bookkeeping variables to store the references of the newly created objects.

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

[16] The full ABs program generated by SmartDepl is available at https://github.com/jacopoMauro/abs_deployer/blob/master/test/output_example.abs.

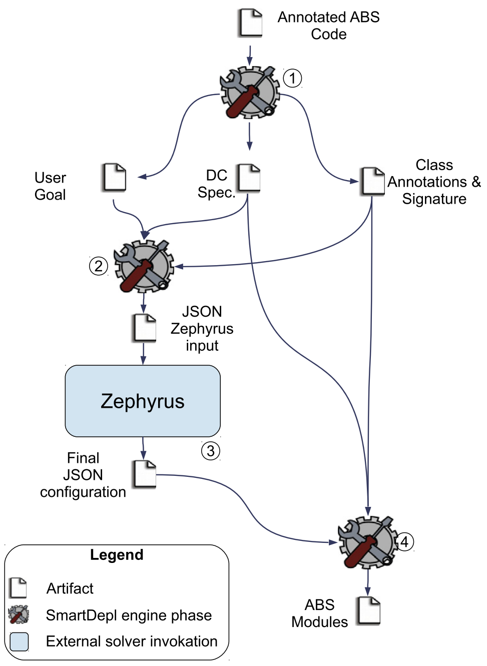


Fig. 3. SmartDepl execution flow.

Finally Lines 12-15 present the invocation of methods for the correct configuration of the system. Line 12 registers the Query Service object to the existing Load Balancer `loadBalancerEndPointObj`. Similarly, Line 13 registers the Query Service to the existing Platform Service object. Line 14 adds the reference of the Query Service to the Deployment Agent that in Line 15 is registered to the existing Deployment Service object.

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

4.5. *Toolchain details*

SmartDepl is open source, available at https://github.com/jacopoMauro/abs_deployer and to increase its portability it can be installed also by using the Docker container technology [53].

The execution flow of SmartDepl is depicted in Fig. 3, representing the different phases with their input and output artifacts. In the first step, SmartDepl parses the annotated ABS program given in input to extract the JSON annotations and the signature of the classes. Intuitively, the following information is retrieved from the parsing: the user goal (a string following the grammar presented in Section 4.3 and given by the user in one ABS annotation), the list of possible DCs that can be used (a JSON representation like the one depicted in Table 1 and given by the user in an ABS annotation), the cost annotations and the signature of every class (JSON internal representation derived from the structure of the ABS program and the user annotations). With this information SmartDepl is able in the second step to generate the input for Zephyrus2[17] [54,8,55], i.e., a configuration optimizer that given the user requirements and a universe of components, computes the optimal configuration satisfying the user needs. This process is quite straightforward since Zephyrus2 supports natively constructs and constraints that mirror those of SmartDepl.

In particular, in Zephyrus2, virtual machines are modeled as locations. Each location has a name, a list of resources that it can provide, and an associated cost. Applications to be deployed on virtual machines are represented as components: black-boxes that expose require- and provide-ports to capture required and provided functionalities respectively. Connections (bindings) from require- to provide-ports model the usage of services.

A detailed recap of the Zephyrus2 tool and its model is outside the scope of this paper (for this we refer the reader to [55]). In the following we will describe how, starting from ABS annotations, SmartDepl generates the Zephyrus2 input and we provide as an example some snippets of code.

DCs are encoded into Zephyrus2 locations and their resources are encoded into the resources offered in the locations. For instance the first two DC types of Table 1 are encoded in Zephyrus2 as follows.

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

```
{ "c4_xlarge_us": {"cost": 209, "num": 5, "payment_interval": 1,
  "resources": {"Cores": 4, "fictional_res": 1, "Memory": 750}},
 "c4_xlarge_eu": {"cost": 209, "num": 5, "payment_interval": 1,
  "resources": {"Cores": 4, "fictional_res": 2, "Memory": 750}},
  ... }}
```

The translation adds two properties, namely `num` and `fictional_res`. The `num` keyword is added to state the maximum amount of that DC types. By default, if not overridden with the annotation `"cloud_provider_DC_availability"`, this default is 5. The property `fictional_res` instead is a unique integer identifier associated to every DC type and added for technical reasons. Indeed, since Zephyrus2 internally uses symmetry breaking constraint to speed up the search, the `fictional_res` identifier is used to avoid Zephyrus2 treating as equal two DC types providing the same resources but having a different name. For example, in the previous case, without the introduction of the `fictional_res` identifier, the two DC types `c4_xlarge_us` and `c4_xlarge_eu` would have been considered equivalent by Zephyrus2 since they were offering the same resources.

SmartDepl also adds a fictional location that simulates the location containing the initial objects.[18] This location is defined as follows.

```
{ "___initial_DC":
 {"num": 1, "cost": 0,
 "resources": {"initial_obj_resource": 1000}}}
```

As far as ABS objects are concerned, their deployment scenarios are mapped into Zephyrus2 component types. For example, the two component types generated by considering the two deployment scenarios of the QueryServiceImpl in Listing 1 are as follows.

```
"staging___QueryServiceImpl": {
  "requires": {"DeploymentAgent": 1},
  "resources": {"Cores": 2, "Memory": 700},
  "provides": [{"num": -1, "ports": ["IQueryService", "Service"]}]},
"live___QueryServiceImpl": {
  "requires": {"DeploymentAgent": 1},
  "resources": {"Cores": 1, "Memory": 300},
  "provides": [{"num": -1, "ports": ["IQueryService", "Service"]}]}
```

The information given in input to Zephyrus2 here is a subset of the information presented in the original annotation where: i) the scenario name has been encoded into the component type name, ii) the required objects are encoded with the `"requires"` keyword, iii) the `"resources"` maps to the class costs, and iv) `"provides"` represents the interface of the object. It is easy to see, e.g., that a `staging___QueryServiceImpl` component type maps an object of class `QueryServiceImpl` deployed in staging modality and therefore consuming 2 `Cores` and 700 units of `Memory`. The `live___QueryServiceImpl` is similar but consumes less memory.

Initial objects are treated as normal objects with the exception that they require the resource `"initial_obj_resource"` that allows them to be deployed only on the location `"___initial_DC"`. Every initial object gets a unique identifier.

Since the structure of the grammar for the user constraints and binding preferences is similar and inspired by the one used by Zephyrus2, the encoding of the constraints and preferences is done by performing a one to one straightforward mapping of the formulas, just mapping DC names (or regular expressions) into location names, and object names into component type names.

For instance, the specification in Listing 2 requiring the presence of the DeploymentServiceImpl where a QueryServiceImpl is deployed is converted as follows.

```
forall ?x in locations : (
  ?x.staging___QueryServiceImpl + ?x.live___QueryServiceImpl > 0 impl
  ?x.default___DeploymentServiceImpl > 0)
```

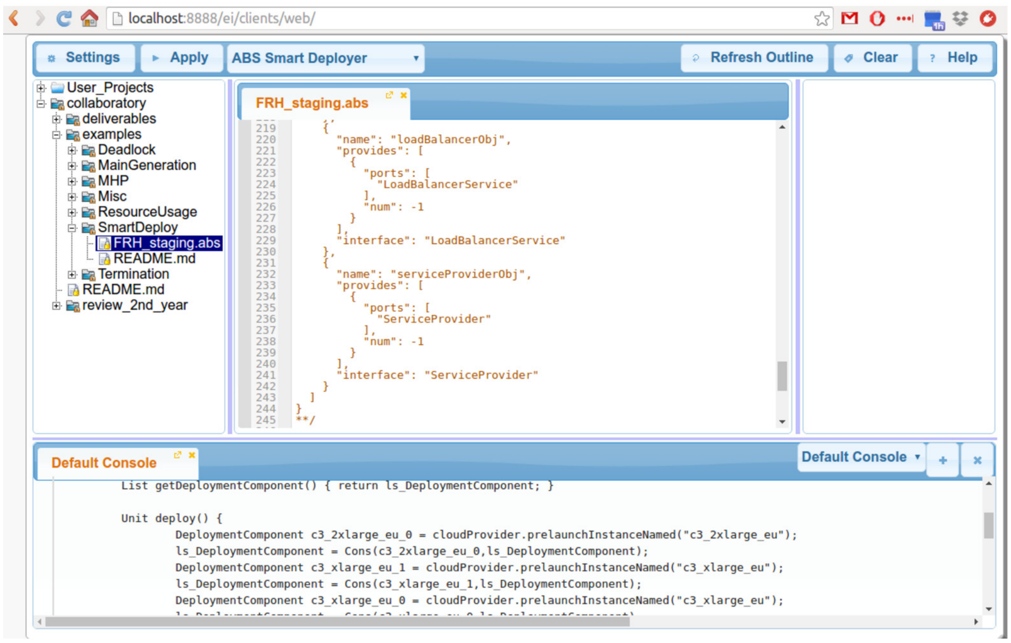
In case initial objects are defined, these are forced to be deployed in only one instance in the `"___initial_DC"` location. For instance, given the initial object `"init_obj"`, the following string is added in the goal specification of Zephyrus2.[19]

```
___initial_DC[0].init_obj= 1
```

When the Zephyrus2 input is generated, SmartDepl runs it (step 3 of Fig. 3). As standard practice, the execution first calls the configurator that returns an abstract configuration that states for every location the number of component types deployed in that location. This information is used in the second Zephyrus2 execution phase for the biding optimization that returns the final configuration listing also all the connections between the different components. The final configuration is a JSON file that, intuitively, can be seen as a directed graph with labeled nodes where the nodes are the components, the

18 By default, the maximal number of initial nodes is 1000.

19 The step by step translation of the original annotations into Zephyrus2 notation and the names mappings can be also visualized for a specific input by running the tool in verbose modality.



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

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

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

Fig. 4. SmartDepl execution within the ABS toolchain IDE.

links are the component dependencies, and the label of the nodes are the locations where the corresponding components are deployed.

SmartDepl parses the optimal configuration returned and it creates the ABS module that generates the corresponding configuration (step 4 of Fig. 3). It is a rather straightforward task, mainly involving the topological sort of the above graphs to decide in which order the components should be deployed. More precisely, the ABS code for the deployment is generated by first creating the DCs of the optimal configuration produced by Zephyrus2. For every location used, the corresponding DC is created by invoking the cloudProvider.prelaunchInstanceNamed method. Then, all the components with their dependencies are considered. The order in which these objects are initialized is obtained by performing a topological sort of the components considering only the dependencies corresponding to class parameters. The objects are created with the **new** constructor and placed into their DC with the [DC: X] annotation where X is the DC assigned to that object. Note that, in case no solution was computed by Zephyrus2 (i.e., the goal was not reachable) or it is impossible to perform a topological sort due to circular dependencies between components, an error is returned. Finally, the code to add the remaining dependencies is generated by invoking the methods specified in the `"methods"` annotation.

As an example, the ABS deployment code for the deploy instructions is available in Listing 5. This code is the one generated when Zephyrus decides to deploy an object DeploymentAgentImpl and an object IQueryService in a m4_large_eu instance. We can first see that the only instance is created by invoking cloudProvider.prelaunchInstanceNamed at Line 2. Then, since the service IQueryService requires a parameter of type DeploymentAgentImpl, the topological sort of the components requires that the DeploymentAgentImpl needs to be deployed before the IQueryService. These objects are indeed created in the corresponding order at Lines 5 and 7 respectively. It is easy to see that at Line 7 the creation of the object of type IQueryService is using as a parameter the object o1 created at Line 5. The remaining dependency to deal with and computed by Zephyrus are due to the fact that the service `IQueryService` when created has to be registered to the objects `loadBalancerEndPointObj` and platformServiceObj that were already existing, and to the object o1. Moreover, also o1 has to be registered to the existing object deploymentServiceObj. These missing dependencies generate the method invocations at Lines 12-15.

To generate the code to undeploy a configuration, the same steps are performed, but in reverse order. First the methods to remove the dependency are invoked (instead of the add methods) and then all the objects and their DC components are deleted with the cloudProvider.shutdownInstance method.

All the steps performed by SmartDepl are polynomial with the exception of the call to Zephyrus2 that solves an NP-hard problem [55]. For this reason, in the general case, there is no guarantee about efficient running times of the SmartDepl. Nevertheless, the experimental validation that we conducted on the Fredhopper Cloud Services case study are satisfactory (see Section 6 for details about the experimental evaluation of the running times).

As a final remark, we would like to remark that, as illustrated in Fig. 4, SmartDepl has also been integrated into the ABS tool chain,[20] an IDE for a collection of tools for writing, inspecting, checking, and analyzing ABS programs.

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

Table 4
Target query processing times for Service Degradation SLA per Customer.

Customer	$\leq$ 200 ms	$\leq$ 500 ms
cust1	99.5%	99.9%
cust2	95%	99%

5. Application to the Fredhopper use case

This section reports on the modeling with SmartDepl of the concrete deployment requirements of the Fredhopper Cloud Services, previously introduced in Section 3. The Fredhopper Cloud Services use case is a suitable candidate to apply and evaluate our techniques for several reasons. It is a challenging case study with complex deployment requirements. The kinds of computing nodes (Amazon AWS based) per customer are known, and since extensive profiling information of the in-production system was available (through many thousands of log files, and millions of queries) the cost of its services can be derived.

SmartDepl was used twice, with two distinct deployment annotations: first, to synthesize the initial static deployment of the entire framework, and second, to synthesize dynamic deployment actions. The generated deployment script to synthesize the initial static deployment is executed once, and realizes the cloud architecture shown in Fig. 2. The dynamic deployment actions mainly concern auto-healing (in case of faults) and scaling in and out instances of the Query Service (including any instances of auxiliary services that the Query Service requires). In contrast to the initial static deployment, dynamic actions are typically executed several times: every time a scaling action must be carried out. The question arises: when and how to scale? The reply is given below, where we describe the integration of SmartDepl in the monitoring framework of the ABS model of the Fredhopper Cloud Services.

5.1. Integration into the monitoring layer

Our technique fully supports elasticity as described above: we integrated SmartDepl into the monitoring framework. The basic idea is to invoke the provisioning script with deployment actions generated by SmartDepl *inside the monitors*. Abstractly, a monitor captures one or more service metrics: each metric is a function that maps an event trace of interactions of customers with the Service APIs to a value. The value indicates the current QoS level of the metric. To ensure a high quality of service, web shops negotiate an aggressive Service Level Agreement (SLA) with Fredhopper. At Fredhopper, the following SLA negotiated with a customer expresses *service degradation* requirements (the exact percentages are negotiable):

> *"Services must maintain 100 queries per second with less than 200 milliseconds respectively 500 ms according to the target percentages in Table 4, ignoring the 2% slowest queries."*

Table 4 shows target values of these metrics for two (anonymized) customers as agreed in the SLA. In general, in our setting, monitors formalize such high-level metrics used in the SLAs (rather than only lower-level metrics such as CPU usage). By comparing the current level of the metrics with the desired target values as agreed in the SLA, the monitor proposes (if needed) scaling suggestions to the cloud engineer. The cloud engineer can then take scaling decisions at the "management level" - the abstraction level of the SLA itself.

When the cloud engineer selects a scaling suggestion, the corresponding provisioning script generated by SmartDepl is invoked and dynamic deployment actions are executed to realize the change. By restricting the scaling actions to those synthesized by SmartDepl, the resulting deployment configuration satisfies all (logical, resource) requirements *by design*.

5.2. Specifying FRH deployments

Since the Fredhopper Cloud Services uses Amazon EC2 "xlarge" and "2xlarge" Compute Optimized instance types (version 3),[21] we used deployment components corresponding to these instances types. For fault tolerance and stability, Fredhopper Cloud Services uses instances in multiple regions in Amazon (regions are geographically separate areas, so even if there is a force majeure in one region, other regions may be unaffected). We model the instance types in different regions as follows: "c3_xlarge_eu", "c3_xlarge_us", "c3_2xlarge_eu", "c3_2xlarge_us" ("eu" refers to a European region, "us" is an American region).

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

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

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

For the correct functioning of the system, a Query Service requires a Deployment Service installed on the same machine. This constraint is expressed as shown in Section 4.3. The requirement that a Service Provider is present on every machine containing a Platform Service is expressed by:

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

Not all services can be freely installed on an arbitrary virtual machine. For instance, to increase resilience, we require that the Load Balancer service runs on a dedicated virtual machine. Listing 3 in section 4.3 shows how this is expressed.

To handle catastrophic failures, the Fredhopper Cloud Services aim to balance the Query Services between the availability zones in regions (see Section 3). This is enforced by constraining the number of the Query Services in the different data centers to be equal. In DRL this is expressed with regular expressions as follows.

```
 (sum ?x in '.*_us1': ?x.QueryServiceImpl['.*']) =
 (sum ?x in '.*_us2': ?x.QueryServiceImpl['.*'])
```

As described in Section 4.3, for performance reasons, the Query Service in Staging mode should be located in the zone of the Platform Service, since Amazon connects instances in the same region with low-latency links. For the European data-center this is expressed by:

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

The previous specifications formalized logical deployment requirements. The next property shows how resource requirements for `QueryService` instances are specified in two workload profiles: a default scenario, and a heavy usage scenario. Tenancy is expressed with the `MaxUse` clause.

```
[Deploy: scenario[Name("DefaultUsage"), MaxUse(1),
                  Cost("CPU", 1), Cost("Memory", 3000),
                  Param("c", User), Param("ds", Req)] ]
[Deploy: scenario[Name("HeavyUsage"), MaxUse(1),
                  Cost("CPU", 2), Cost("Memory", 4500),
                  Param("c", User), Param("ds", Req)] ]
class QueryServiceImpl(DeploymentService ds, Customer c)
    ...
```

Binding preferences To allow the connection of components with components located in the same availability zone or region, SmartDepl introduced the notion of binding preferences. SmartDepl first computes the components for the optimal configuration and then it uses these preferences to compute the final configuration establishing the component connections that maximize the binding preferences.

As an example, the `QueryService` in the US region must be connected to the `LoadBalancerEndPoint` deployed in the same region. SmartDepl enforces this with the following preference.

```
sum ?x of type QueryServiceImpl['.*'] in '.*_us.' :
 forall ?y of type LoadBalancerEndPointImpl
  in '.*_us.' : ?x used by ?y
```

This preference requires to maximize the number of connections between every `QueryService` deployed in US region and every `LoadBalancerEndPoint` in that same region. As a consequence, all the `QueryService` in the US region will be connected to all the `LoadBalancerEndPoint` in the same region.

Binding preferences also make it possible to avoid establishing connections between certain objects, or add connections to existing objects instead of newly created instances.

Installation actions To successfully deploy new service instances, certain installation actions should be executed. For example, whenever a new `QueryService` is added, it should be added to the appropriate load balancers (all load balancers in the same region as the new query service) through an `add(Service)` method. It should then be announced to the platform through an `addServiceInstance` method. The last step is to finalize the deployment of the new query service, by calling the `install` method on the deployment service.

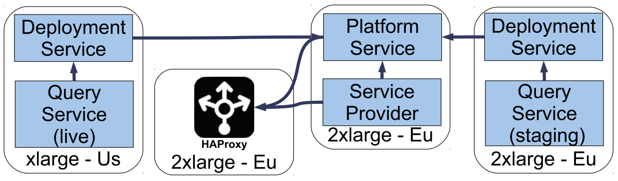


Fig. 5. Example of automatic objects allocation to deployment components.

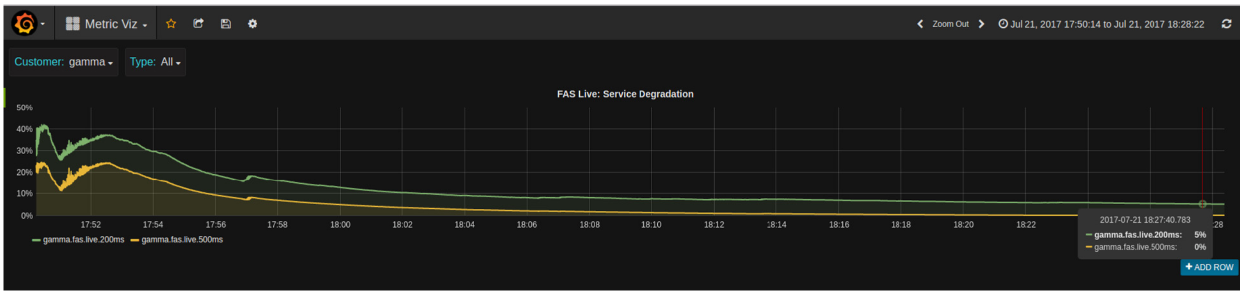


Fig. 6. Auto-healing and scaling.

The order of these actions are important and can be specified as follows:

```
"add_method_priorities":[
  { "class":"LoadBalancerEndPointImpl",
    "method":"add" },
  { "class":"PlatformServiceImpl",
    "method":"addServiceInstance" },
  { "class":"DeploymentAgentImpl",
    "method":"install" }
]
```

5.3. Results

From the above specifications, SmartDepl computes the initial configuration in Fig. 5, which minimizes the total costs of the deployment per time unit. This is calculated based on the cost per interval of each kind of VM, and the length of that interval. The length of the interval (in terms of ABS time units) can be specified in the JSON file for each VM type individually using the payment_interval keyword. For example, if cost is 100 and payment_interval is 2, then every 2 time units that the machine is in a running state, the costs are increased by 100 (and that VM contributes 50 every time unit to the total cost of the deployment). SmartDepl deploys the Load Balancer, Platform Service and one staging Query Service on three “2xlarge” instances in Europe, and deploys a live Query service on an “xlarge” instance in US.

Simulation After the initial deployment, the Cloud engineers of Fredhopper Cloud Services rely on feedback provided by monitors to decide if more Query Services in live mode are needed. We simulated real-world scenario’s (by driving the simulation with query data from several GB of log files[22]), using the code generated by SmartDepl to instantiate service instances and resources appropriately. This helped to calibrate the model and validate the results through a comparison with the production environment.

Fig. 7 and 8 show several of the metrics for a single customer used to determine subsequent scaling actions: number of queries per second for the customer, number of requests in process, and CPU usage (Fig. 7) and load, memory usage and swap space usage (Fig. 8). Both the production system and the ABS model use the Grafana framework[23] to visualize the metrics. The timescale in the figures is 1 day, but this can be adjusted to see trends over longer periods, or zoom in on a short period. The figures show that the number of queries served per second (qps, first graph of Fig. 7) is relatively high and the requests (Fig. 7, second graph) are fairly low, so requests are not queuing. Furthermore the CPU usage (Fig. 7, third graph) and memory consumption with small swap space used (Fig. 8, second and third graphs) look healthy. Hence, no scaling is needed.

If we would have needed to scale out, *two* Query Service instances are added: one in an EU region, and one in a US region for balancing across regions. In contrast, if there is unnecessary overcapacity, the most recent ones can be shut down.

[22] The simulation can also be driven without using log files by forwarding live queries from the production environment to the ABS model.
[23] https://grafana.com/.

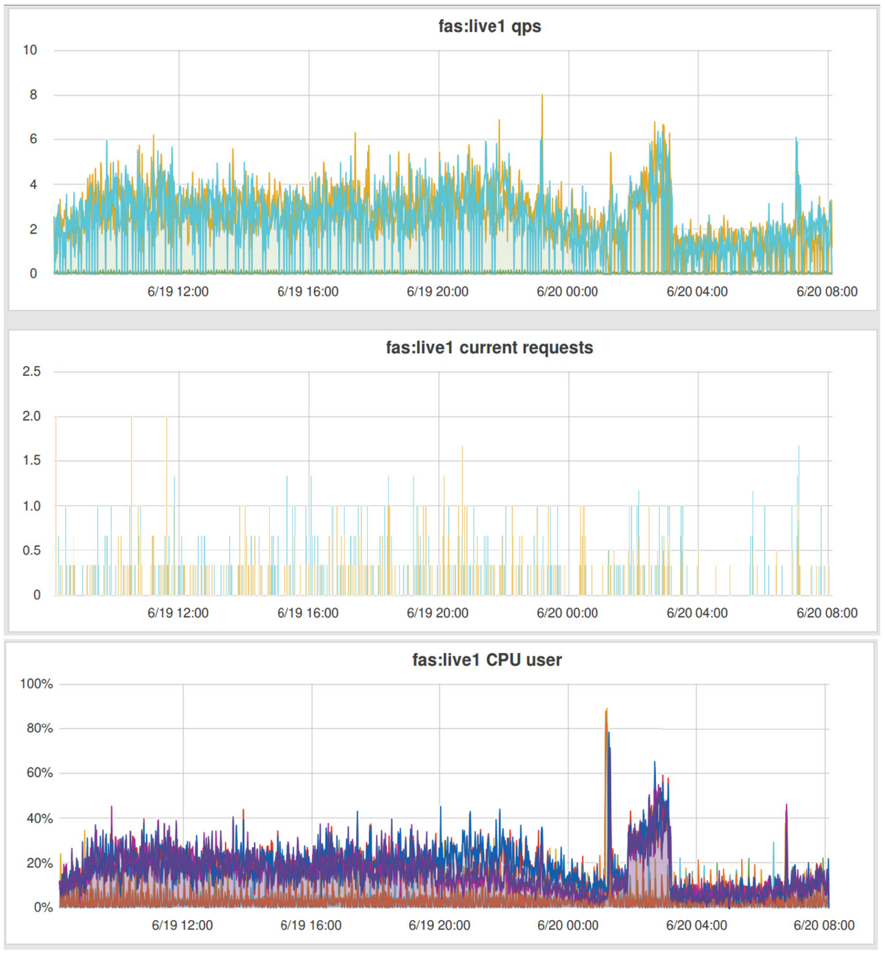


Fig. 7. Metrics graphed over a single day for a customer.

Fig. 6 shows the simulation of a second scenario. Real-world log files from a customer cust2 were taken and replayed with an auxiliary tool logreplay[24] against the ABS model's LoadBalancer endpoints (which then forward the queries in round-robin fashion to a QueryService instance). All service instances and resources were deployed with SmartDepl. The figure shows the values over time of the two 200 ms and 500 ms Service Degradation metrics from the SLA with cust2 from Table 4. The graph is generated in "real-time" (there is a ten-second delay to publish metric values to an external database, and visualize the metrics with Grafana). The metric with the higher values shows the percentage of queries slower than 500 ms, the lower line is the percentage of queries slower than 200 ms. As the graph shows, after the initial phase, the numbers stabilize and satisfy the SLA. The chaotic initial phase is caused by the fact that at the start, the service instances are initializing and cannot process queries until initialization finishes, thus temporarily violating the target QoS values from the SLA.

In the visualized scenario, a QueryService instance crashed at the very beginning. The crash is simulated by exposing the undeploy method synthesized by SmartDepl as a method callable over HTTP, and consequently invoking undeploy. The crash leads to a very high degradation initially. After ≈ 1 minute, the degradation monitor detects this and auto-heals by deploying a new QueryService instance. The VM is started and the necessary services are installed by the SmartDepl provisioning script for scaling QueryService at around 17:57. Seemingly strange, this causes initially the degradation to *increase*! The reason is that although the service instance is ready to accept requests, performance is sub-optimal in the beginning due to initialization procedures taking place simultaneously. This still does not prove to be enough to meet the desired target degradation requirements (the table allows 5% queries slower than 200 ms, whereas the graph remains stable at around 9% starting from 18:04). The degradation monitor detects this and suggests to scale out using the SmartDepl query service deployer.

[24] https://github.com/abstools/logreplay.

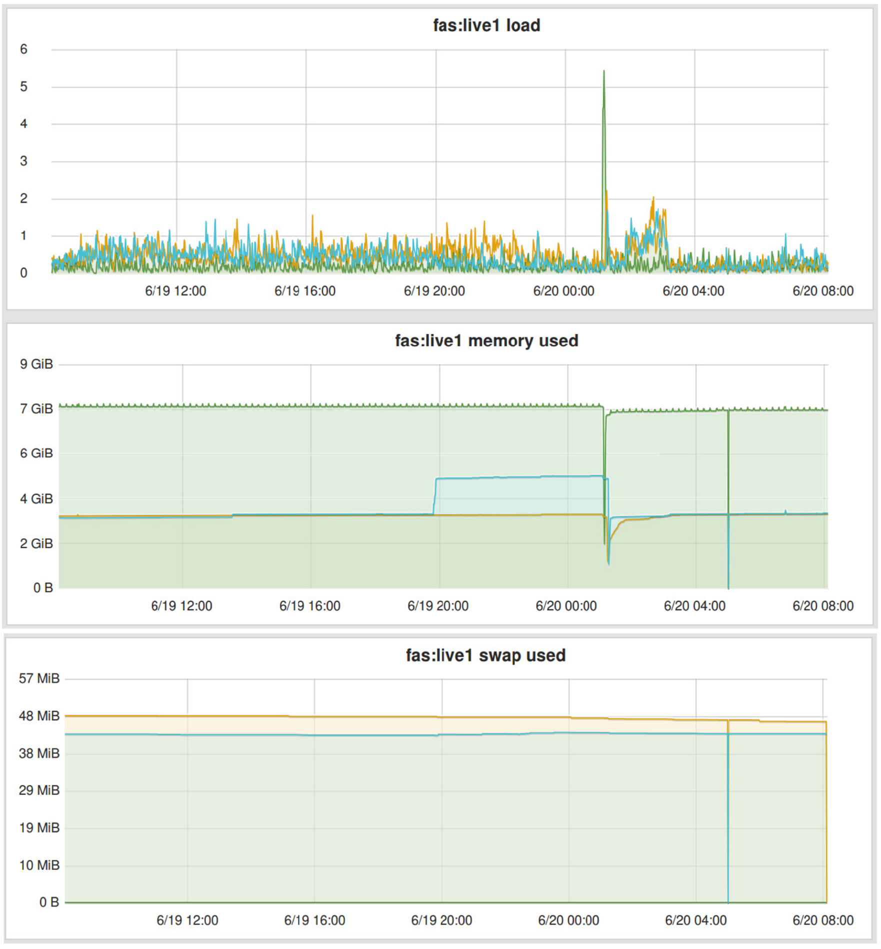


Fig. 8. Metrics graphed over a single day for a customer.

In general, since ABS offers an API that allows to invoke ABS methods over HTTP,[25] one can also forward *in real-time* information about the run-time deployment model (including resource failures). In this way, the ABS model may take also runtime deployment information into account, and can react with appropriate scaling suggestions accordingly.

Either a scaling strategy can be implemented that automatically selects the suggested scaling actions (a very simple strategy is to approve of all suggestions, this is sufficient for auto-healing), or the cloud engineer can select the actions he/she desires manually, based on their domain knowledge and interpretation of the monitored metrics. The simple auto-healing strategy that accepts all suggestions then deploys a second new query service instance at 18:14 and the degradation metrics converge to the desired target values. The final configuration, obtained through first deploying the initial configuration and then twice scaling instances with a AddQueryDeployer corresponds exactly with the deployment configuration used for the in-production system for that customer (for the in-production system, the configuration was obtained manually through the actions taken by the cloud engineers).

To compare the existing work-flow at Fredhopper and the effect of integrating SmartDeploy in this work-flow, we discussed with the operations team their experience. The Cloud operations team currently decides manually how and when to scale. As Fredhopper has very aggressive SLAs, the team is typically conservative with scaling instances out, leading to potential over-spending. The ability of SmartDepl to deploy in the programming language (ABS) itself allows to leverage the extensive tool-supported analyses available for ABS, including logic-based modular verification [4], deadlock detection [5], cost analysis [6] and run-time monitoring [56] (see [57] for an overview of most tools). For example, by using monitors to track the quality of services, SmartDepl allows Ops to reason on the scaling decisions and their impact on the SLA agreed with the customers [58].

[25] https://abs-models.org/manual/#-the-model-api.

The operations team currently use ad-hoc scripts to configure newly added or removed service instances, and these scripts are specific to the infrastructure provider. Thus the question arose: is it possible to find a more flexible and systematic approach that uses *pluggable* infrastructure providers?

This requires first introducing a generic API of an abstract deployment layer that captures the commonalities of different infrastructure providers, and allows plugging in (loading) infrastructure provider-specific information, such as the machine types and their resource properties. SmartDepl provides the required basis for such an approach. From the virtual machine descriptions in JSON, SmartDepl generates code that imports the different machine types into a *generic* Cloud API offered by ABS for managing virtual resources. The provisioning generated scripts by SmartDepl launch, terminate and manage the life-cycle of the required virtual machines through this generic Cloud API (optimizing the cost of the machines). Finally, SmartDepl deploys (installs and configures) the service instances on the virtual machines in such a way that respects all the deployment requirements. The provisioning scripts can be executed (in other words, they form executable code) by choosing an implementation of the generic Cloud API for the desired infrastructure provider. This approach allows SmartDepl to seamlessly switch between different infrastructure providers by leveraging the abstract Cloud API layer - simply plug in an instance of the cloud API implementation of another provider. It even enables usage and analysis of multiple infrastructure providers in the same system, i.e. using a mixture of Amazon AWS instances and Microsoft Azure instances to deploy services on.

The ABS model used with all the annotations and specifications and an example of generated code are available at https://github.com/jacopoMauro/abs_deployer/tree/master/test.

6. SmartDepl solver running times

In this section we present an evaluation of the scaling performance of the SmartDepl solver. As previously stated, to generate the code, SmartDepl relies on Zephyrus2 that solves an NP-hard problem. Due to the nature of this problem, SmartDepl does not provide any guarantee on the running times that in the worst case may be exponential on the size of the input. It is however natural to wonder what are the running times of SmartDepl for normal instances. Unfortunately, as also remarked in [54], there are no standard benchmarks that can be used for the optimization of application deployment. For this reason, in this work we will try to evaluate how good SmartDepl scales by measuring its running times using the real-world Fredhopper use case as a specific benchmark.

In particular, we performed two kinds of scaling experiments. In the first experiment, dubbed `ScaleInitial` we used SmartDepl to generate the code to deploy the initial configuration by varying the number of DC that SmartDepl is allowed to use. By default, for every kind of DC, SmartDepl may use up to 5 DC instances of that type for every deploy method invocation.[26] This number has an impact on the performance since the more DC components can be used in one deploy call, the larger is the search space to check in order to find the optimal deployment solution. SmartDepl allows the customization of the number of DC for every type of DC by using the keyword cloud_provider_DC_availability and specifying for the interested DC types the number of DC that can be used in a deploy invocation. For instance the following addition in a SmartDepl annotation would allow to use only 2 DC of type c4_2xlarge_eu.

```
"cloud_provider_DC_availability":{
  "c4_2xlarge_eu" : 2}
```

We tested SmartDepl by varying the number of allowed DC between 2 (the minimal number in order to have an optimal solution) to 20. Since there were 12 different types of DC this corresponds in considering from 24 to 240 different DC that can potentially be created at every deploy method invocation.

In the second experiment, dubbed `ScaleQueries`, we use instead SmartDepl to generate the code to add a Query service in live mode by varying the number of Query services deployed in the European and US regions by varying the goal specification. We run the experiment by varying the Query services required in the European region between 0 and 10, and by varying the Query services required in the US between 0 and 20. These are the maximal numbers of Query Services that can be deployed considering the default number of DC per DC type of SmartDepl.

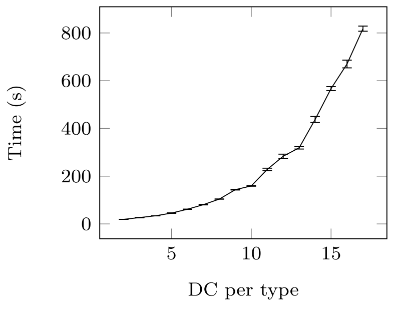
Since SmartDepl has been developed to be deployable on a cloud infrastructure, we validated it by running it on an OpenStack cloud. We used a virtual machine requiring 8 GB of RAM, 4 cores, and running the Ubuntu 16.04 operating system. We repeated every experiment 5 times reporting the average time and the absolute error by considering the output of the Linux time command. For every experiment we used a time limit of 900 s. Since the Zephyrus2 tool supports three different solvers, namely the constraint solver Gecode, the lazy constraint solver Chuffed, and the SMT solver Z3 we have run all the experiments considering these three execution modalities.

The results of the `ScaleInitial` experiments are presented in Table 5. Times are given in seconds, rounded to the second decimal place. It is immediately clear that the best solver to use while executing Zephyrus2 is Chuffed. While for the smallest instance Gecode and SMT are faster, as soon as more DC are allowed to be used we notice that Gecode does not scale while the Z3 scales worse than Chuffed. SmartDepl takes 21.95 seconds to generate the initial deployment code when

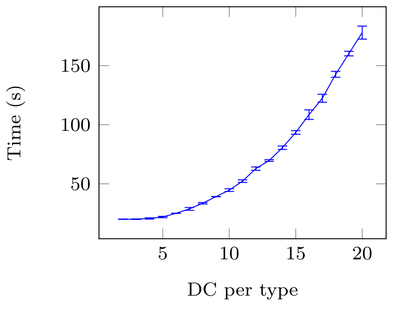
[26] Note that the number of times the deploy method is invoked can be unbounded. Hence, the number of DCs that can be used can be unbounded, but every deployment invocation can create only a bounded number of DCs.

Table 5
Average running times for the `ScaleInitial` experiment (Timeout = 900 s).

DC per type	Z3 (s)	Chuffed (s)	Gecode (s)
2	19.09 ± 0.09	20.08 ± 0.13	13.05 ± 0.16
3	26.60 ± 0.54	20.10 ± 0.18	246.46 ± 2.09
4	34.06 ± 0.07	20.70 ± 0.53	Timeout
5	45.36 ± 1.06	21.95 ± 0.58	Timeout
6	61.57 ± 0.55	25.09 ± 0.09	Timeout
7	80.69 ± 1.00	28.80 ± 1.14	Timeout
8	104.12 ± 0.35	33.56 ± 0.64	Timeout
9	143.46 ± 1.37	39.13 ± 0.13	Timeout
10	159.18 ± 1.36	44.69 ± 1.15	Timeout
11	228.28 ± 5.14	52.28 ± 1.07	Timeout
12	283.50 ± 8.77	62.86 ± 1.44	Timeout
13	318.85 ± 5.13	69.70 ± 0.75	Timeout
14	437.46 ± 12.60	80.56 ± 1.46	Timeout
15	566.91 ± 8.30	93.49 ± 1.56	Timeout
16	669.99 ± 16.21	108.52 ± 4.06	Timeout
17	818.04 ± 10.76	122.37 ± 3.39	Timeout
18	Timeout	142.64 ± 2.49	Timeout
19	Timeout	160.19 ± 1.94	Timeout
20	Timeout	177.94 ± 5.51	Timeout



(a) Z3 results.



(b) Chuffed results.

Fig. 9. `ScaleInitial` results for Z3 and Chuffed.

the default 5 number of DC are allowed per DC type. Within 1 minute SmartDepl can generate the initial configuration considering up to 132 DC (11 DC per DC type). Considering that usually requiring a virtual machine on a cloud can take more than 5 minutes, we believe that such a performance is good enough for the day to day deployment tasks faced by a small/medium organization.

Clearly, running SmartDepl on the cloud may introduce more variability on the running times due to the fact that computational resources may be shared with other users. Since one of the simplest estimates of the uncertainty is the range of the performed measures (i.e., the difference between the highest and the lowest measure), for every test we consider as absolute error half of the range of the 5 repetitions. The absolute error for the Z3 and Chuffed approaches is presented in Fig. 9 using, as is customary, error bars.[27] The relative error was never greater than 4% and we believe this is more than an acceptable price to pay to be allowed to use a non-dedicated cluster to run SmartDepl.

We would like to note that the times presented here do not include only the running time of Zephyrus2 but also the parsing of the ABS file, the process of the annotations, and the printing of the ABS code. Clearly the length of the ABS program may have an influence on the performance but the parsing activities, being bounded polynomially in time, do not present particular challenges and their require a time that is negligible w.r.t. the running times taken by running Zephyrus2.

As far as the `ScaleQuery` experiments are concerned, Fig. 10 presents the results by using 3D plots having as x and y axis the number of Query Services required to be deployed in the European and US regions while the z axis presents the average running time. We indicate with a blue dot on the plane containing the x and y axis, the coordinates for which the solver timeouts.

From the plots it is easy to see that, similarly to what happened for the `ScaleInitial` experiments, the best solver to use is Chuffed able to solve all the instances in less than 215.56 seconds. As before, due to the fact that the computation is performed at compile time, we believe that this performance is good enough for the day to day deployment tasks faced by a small/medium organization.

[27] Due to the fact that Gecode timeouts already at the third instance, we did not consider the plotting of the errors for this approach.

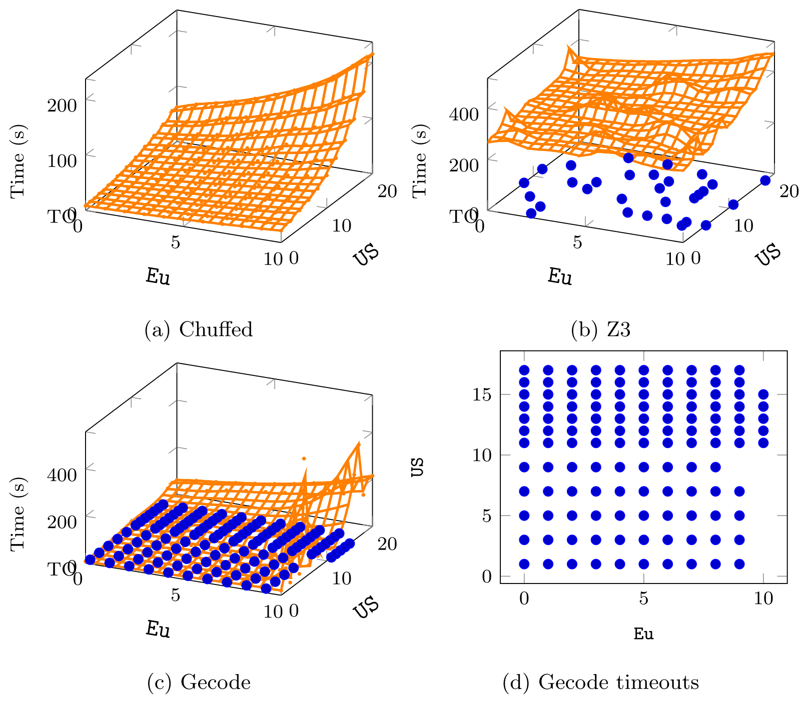


Fig. 10. `ScaleQuery` results (large blue points indicates timeouts).

The Z3 solver timeouts for few times, and it is in average far slower than Chuffed. When it is able to solve an instance it was able to solve it in less than 472 seconds, otherwise it timeouts. It is not clear why for certain instances Z3 timeouts. As it often happens when dealing with NP-hard problems, our conjecture is that the search heuristics for those instances lead the exploration of a non-promising search space that does not allow a lot of pruning.

Gecode instead for some few instances has a performance comparable with Chuffed but timeouts for most of the instances. Even for simple instances, as can be seen from Fig. 10d detailing when Gecode timeouts, Gecode is not able to produce the optimal deployment code in less than 900 seconds. This happens in particular when the number of Query Services required in the US region is odd. We believe that this is due to the fact that one of the constraints of the specification was to divide equally the query services between the data centers in the US region. Since there are two areas in the US, this means that even if we required an odd number of Query Services in live mode a valid initial configuration will still require an even number of Query Services (in live or staging deployment modalities) to be deployed in the US region. Chuffed, that is using a different search heuristic based also on learning, was able to prove optimality quickly while the heuristic employed by the Gecode solver is not effective in this case for reducing the search space.

When the number of the required Query Services was greater than the resources that could be used (i.e., > 10 for Europe or > 20 for the US region), SmartDepl communicates the impossibility of finding the solution in only few seconds. This was due to the fact that the detection of the unsatisfiability of the user constraints was almost instantaneous for all the three solvers used by Zephyrus2.

As far as the variability of the runtime is concerned, the relative error by using Chuffed was less than 5%. Unfortunately for Z3 (resp. Gecode) we experienced for 23 (resp. 5) instances a relative error superior to 10%. In case of Z3 the maximal relative error was close to 62%. We believe that partially this is due to the fact that the cloud infrastructure was possibly shared with other users, but probably the big relative error is due to the non deterministic nature of the SMT solver that could behave very differently according to the random seed chosen to regulate its internal decisions. Investigating this issue further is beyond the scope of this paper, since Chuffed did not present this problem and was better than both Z3 and Gecode.

7. Conclusions

We presented an extension of the ABS specification language that supports modeling cloud application deployment in a declarative manner: the programmer specifies deployment constraints, and a solver synthesizes ABS classes with methods that execute deployment actions to reach an optimal deployment configuration that satisfies the constraints. We are not aware of other approaches that used formal tools to optimize the deployment of applications at the modeling level. Our approach, which is inspired by [8] and significantly improves our initial work [14], can be easily applied to any other object-oriented language that offers primitives for the acquisition and release of computing resources.

In the light of this positive validation, obtained by means of the modeling and analysis of the Fredhopper Cloud Services, we can conclude that our approach was successful for at least three main factors:

- the reasoning about application deployment at the modeling level,
- the possibility to express with a domain-specific language the deployment constraints (that are usually only implicit in the best practices of the operations experts),
- and the automatic synthesis of optimal deployments.

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

Another limitation of the current version of SmartDepl is that when no solution exists that satisfy all the user requirements, the user is only notified about it. We are planning to extend SmartDepl with explanation based mechanism such as [59] to help the users to identify what are the conflicting constraints.

Declaration of Competing Interest

We have no competing interests.

References

[1] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.
[2] MODAClouds and DICE projects, http://multiclouddevops.com/.
[3] Abstract behavioral specification language, http://www.abs-models.com/.
[4] C.C. Din, R. Bubel, R. Hähnle, Key-abs: a deductive verification tool for the concurrent modelling language ABS, in: CADE, 2015.
[5] E. Giachino, C. Laneve, M. Lienhardt, A framework for deadlock detection in core ABS, CoRR, http://arxiv.org/abs/1511.04926.
[6] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, G. Román-Díez, SACO: Static Analyzer for Concurrent Objects, in: ETAPS, 2014.
[7] J.A. Hewson, P. Anderson, A.D. Gordon, A declarative approach to automated configuration, in: LISA, 2012.
[8] R.D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, A. Agahi, Automated synthesis and deployment of cloud applications, in: ASE, 2014.
[9] C. Qu, R.N. Calheiros, R. Buyya, Auto-scaling web applications in clouds: a taxonomy and survey, ACM Comput. Surv. 51 (4) (2018) 73:1–73:33, https://doi.org/10.1145/3148149.
[10] K. Hightower, B. Burns, J. Beda, Kubernetes: Up and Running Dive into the Future of Infrastructure, 1st edition, O'Reilly Media, Inc., 2017.
[11] S. Dutta, S. Gera, A. Verma, B. Viswanathan, SmartScale: automatic application scaling in enterprise clouds, in: 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24–29, 2012, IEEE Computer Society, 2012, pp. 221–228.
[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, L. Zhang, Modeling the impact of workload on cloud resource scaling, in: 26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22–24, 2014, IEEE Computer Society, 2014, pp. 310–317.
[13] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: ASPLOS, ACM, 2019, pp. 3–18.
[14] S. de Gouw, M. Lienhardt, J. Mauro, B. Nobakht, G. Zavattaro, On the integration of automatic deployment into the ABS modeling language, in: ESOCC, 2015.
[15] S. de Gouw, J. Mauro, B. Nobakht, G. Zavattaro, Declarative elasticity in ABS, in: ESOCC, in: LNCS, vol. 9846, Springer, 2016, pp. 118–134.
[16] Google App Engine, https://developers.google.com/appengine/.
[17] Microsoft Azure, http://azure.microsoft.com.
[18] DevOps, http://devops.com/.
[19] 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.
[20] OASIS, Organization for the Advancement of Structured Information Standards (OASIS), https://www.oasis-open.org.
[21] Business Process Model and Notation (BPMN), http://www.bpmn.org/.
[22] Business Process Execution Language (BPEL), https://www.oasis-open.org/committees/wsbpel/.
[23] Juju, devops distilled, https://juju.ubuntu.com/.
[24] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann, A systematic review of cloud modeling languages, ACM Comput. Surv. 51 (1) (2018) 22:1–22:38, https://doi.org/10.1145/3150227.
[25] MARTE profile, https://www.omg.org/omgmarte/.
[26] Object Constraint Language (OCL), https://www.omg.org/spec/OCL/.
[27] R.D. Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Aeolus: a component model for the cloud, Inf. Comput. 239 (2014) 100–121.
[28] T.A. Lascu, J. Mauro, G. Zavattaro, Automatic deployment of component-based applications, Sci. Comput. Program. 113 (2015) 261–284.
[29] A. Brogi, A. Canciani, J. Soldani, Fault-aware application management protocols, in: ESOCC, in: LNCS, vol. 9846, Springer, 2016, pp. 219–234.
[30] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, N.D. Palma, Reliable self-deployment of distributed cloud applications, Softw. Pract. Exp. 47 (1) (2017) 3–20, https://doi.org/10.1002/spe.2400.
[31] M. Burgess, A site configuration engine, Computing Systems (2).
[32] L. Kanies, Puppet: next-generation configuration management, ;Login: 31 (1) (2006).
[33] Puppet Labs, Marionette collective, http://docs.puppetlabs.com/mcollective/.
[34] Opscode, Chef, http://www.opscode.com/chef/.

[35] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: a deployment management system, in: PLDI, 2012.
[36] J.Z. Li, C.M. Woodside, J.W. Chinneck, M. Litoiu, Cloudopt: multi-goal optimization of application deployments across a cloud, in: CNSM, IEEE, 2011, pp. 1–9.
[37] J.L. Lucas-Simarro, R. Moreno-Vozmediano, R.S. Montero, I.M. Llorente, Scheduling strategies for optimal service deployment across multiple clouds, Future Gener. Comput. Syst. 29 (6) (2013) 1431–1441.
[38] L. Thai, A. Barker, B. Varghese, O. Akgun, I. Miguel, Optimal deployment of geographically distributed workflow engines on the cloud, in: IEEE CloudCom, IEEE Computer Society, 2014, pp. 811–816.
[39] W. Li, P. Svärd, J. Tordsson, E. Elmroth, Cost-optimal cloud service placement under dynamic pricing schemes, in: IEEE/ACM UCC, IEEE Computer Society, 2013, pp. 187–194.
[40] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: a systematic literature review, in: Software Engineering 2014, in: LNI, GI, vol. 227, 2014, pp. 77–78.
[41] X. Etchevers, T. Coupaye, F. Boyer, N.D. Palma, Self-configuration of distributed applications in the cloud, in: IEEE CLOUD, IEEE, 2011, pp. 668–675.
[42] G. Salaün, X. Etchevers, N.D. Palma, F. Boyer, T. Coupaye, Verification of a self-configuration protocol for distributed applications in the cloud, in: Assurances for Self-Adaptive Systems, in: LNCS, vol. 7740, Springer, 2013, pp. 60–79.
[43] DMTF (Distributed Management Task Force), Open virtualization format specification version 2.0.1, http://dmtf.org/sites/default/files/standards/documents/DSP0243_2.0.1.pdf.
[44] HashiCorp, Terraform, https://terraform.io/.
[45] R. Zabolotnyi, P. Leitner, W. Hummer, S. Dustdar, JCloudScale: closing the gap between IaaS and PaaS, ACM Trans. Internet Technol. 15 (3) (2015) 10.
[46] Apache Software Foundation, Apache Brooklyn, https://brooklyn.incubator.apache.org/.
[47] OASIS, Cloud application management for platforms, http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.
[48] M. Caballer, C. de Alfonso, F. Alvarruiz, G. Moltó, EC3: elastic cloud computing cluster, J. Comput. Syst. Sci. 79 (8) (2013) 1341–1351, https://doi.org/10.1016/j.jcss.2013.06.005.
[49] Kubernetes Authors, Kubernetes, https://kubernetes.io/.
[50] P. Goldsack, J. Guijarro, S. Loughran, A.N. Coles, A. Farrell, A. Lain, P. Murray, P. Toft, The SmartFrog configuration management framework, Oper. Syst. Rev. 43 (1) (2009) 16–25.
[51] J. Mirkovic, T. Faber, P. Hsieh, G. Malaiyandisamy, R. Malaviya, DADL: Distributed Application Description Language, USC/ISI Technical Report.
[52] Cloud Foundry, http://cloudfoundry.org/.
[53] Docker Inc, Docker, https://www.docker.com/.
[54] E. Ábrahám, F. Corzilius, E.B. Johnsen, G. Kremer, J. Mauro, Zephyrus2: on the fly deployment optimization using SMT and CP technologies, in: SETTA, in: LNCS, vol. 9984, 2016, pp. 229–245.
[55] R.D. Cosmo, M. Lienhardt, J. Mauro, S. Zacchiroli, G. Zavattaro, J. Zwolakowski, Automatic application deployment in the cloud: from practice to theory and back, in: CONCUR, 2015.
[56] F.S. de Boer, S. de Gouw, P.Y.H. Wong, Run-time verification of coboxes, in: Software Engineering and Formal Methods - Proceedings of the 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013, 2013, pp. 259–273.
[57] P.Y.H. Wong, R. Bubel, F.S. de Boer, M. Gómez-Zamalloa, S. de Gouw, R. Hähnle, K. Meinke, M.A. Sindhu, Testing abstract behavioral specifications, Int. J. Softw. Tools Technol. Transf. 17 (1) (2015) 107–119, https://doi.org/10.1007/s10009-014-0301-x.
[58] N. Bezirgiannis, F.S. de Boer, S. de Gouw, Human-in-the-loop simulation of cloud services, in: ESOCC, in: Lecture Notes in Computer Science, vol. 10465, Springer, 2017, pp. 143–158.
[59] O. Guthmann, O. Strichman, A. Trostanetski, Minimal unsatisfiable core extraction for SMT, in: FMCAD, IEEE, 2016, pp. 57–64.