



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

Deliverable D5.3 Envisage Work Flow

Date of document: T0+24



Start date of the project: **1st October 2013**

Duration: **36 months**

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

Final version

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

Executive Summary:

Envisage Work Flow

This document summarizes deliverable D5.3 of project FP7-610582 (**Envisage**), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at <http://www.envisage-project.eu>.

Deliverable D5.3 reports on how **Envisage** technologies may be used in software development processes by suggesting work flows for using the ABS modeling language and associated tools. Two different and complementary work flows are discussed in the deliverable and related to existing software development processes.

List of Authors

David Costa (FRH)
Einar Broch Johnsen (UIO)
Stijn de Gouw (FRH)
Behrooz Nobakht (FRH)

Contents

1	Introduction	4
1.1	Model-driven Engineering of Virtualized Services	4
1.2	Work Flows	6
1.3	Dissemination of the Proposed Methodology	6
2	Background on Envisage Outcomes	7
2.1	Approach.	7
2.2	Overview of the ABS Modeling Language.	7
2.3	Overview of the Analysis Tools Developed for ABS.	8
2.4	Overview of the Collaboratory.	8
3	A Model-Driven Development Process with ABS	9
3.1	Requirement Analysis	10
3.2	Design	10
3.3	Implementation	10
3.4	Testing	10
3.5	Evolution	10
4	A DevOps Enabled Work Flow for ABS	12
4.1	Creating the ABS Model	12
4.2	Automated Tests	13
4.3	Automated Configuration	14
4.4	Continuous Monitoring and Feedback	14
4.5	Collaborative Development	15
4.6	Continuous Integration	16
	Bibliography	16
	Glossary	19
A	Engineering Virtualized Services	20
B	Designing Resource-Aware Cloud Applications	26

Chapter 1

Introduction

Today, deployment decisions are typically made late in the software development process, after design, implementation, and testing phases [4]. Virtualized services deployed on the cloud often need to adapt to different deployment scenarios, e.g., depending on the agreed service level or on demand. To avoid a potentially costly redesign of a service because of unnecessarily high operational costs or bad scalability, the **Envisage** project aims to shift deployment decisions from the service deployment phase into the service design phase of the software development by integrating deployment decisions into the service models of the design phase. To enable this shift, the vision of **Envisage** is that requirements which have so far only been informally expressed, such as service-level agreements, need to be integrated into an executable formal modeling language that also captures deployment aspects, which are normally confined to the underlying infrastructure. In this deliverable, we discuss two approaches to capitalize from this vision in the engineering of virtualized services.

This deliverable discusses how the outcomes of the **Envisage** project can contribute to the model-driven engineering of virtualized services. The deliverable first briefly discusses the model-driven engineering of virtualized services and surveys the modeling and technology solutions proposed by the **Envisage** project. The deliverable then places these solutions in the context of work flows of software engineers. The aim of this deliverable is not to discuss the details of the proposed solutions, which are detailed in other project deliverables, but to place them in the context of the working practices of software engineers. We consider two different kinds of work flow: (1) a model-driven development process and (2) a DevOps enabled work flow.

1.1 Model-driven Engineering of Virtualized Services

Conceptually, a deployed service on the cloud consists of more than just the code defining its functionality. The development decisions needed to deliver the service to customers also involve the *service-level agreements* associated with the service and the *provisioning of resources* required to run the service. Figure 1.1 illustrates these three layers of a deployed cloud service. **Envisage** aims to support the developer in making decisions covering the spectrum from design to deployment.

The Approach of Envisage. Figure 1.2 illustrates the approach taken by **Envisage** to enable model-based analysis to address the three layers of deployed services. Resource-awareness is obtained by allowing the services to interact with the cloud provisioning layer through the “Cloud API”, which can be adjusted to offer virtual machine instances with various profiles and provides resource accounting over time for rapid prototyping of resource management decisions. As shown in the figure, the framework proposed by **Envisage** does not aim at covering the Legal Contract Layer, but focusses on non-functional properties related to deployment decisions and (traditional) correctness properties for the functional behavior of the services.

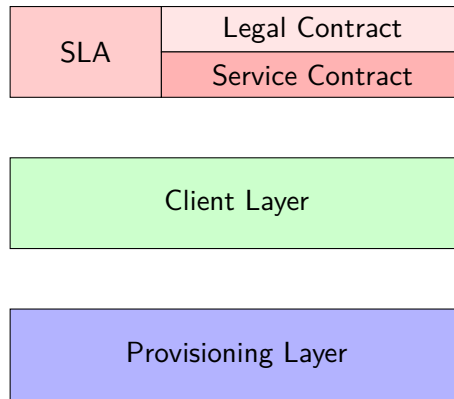


Figure 1.1: Conceptual parts of a deployed cloud service

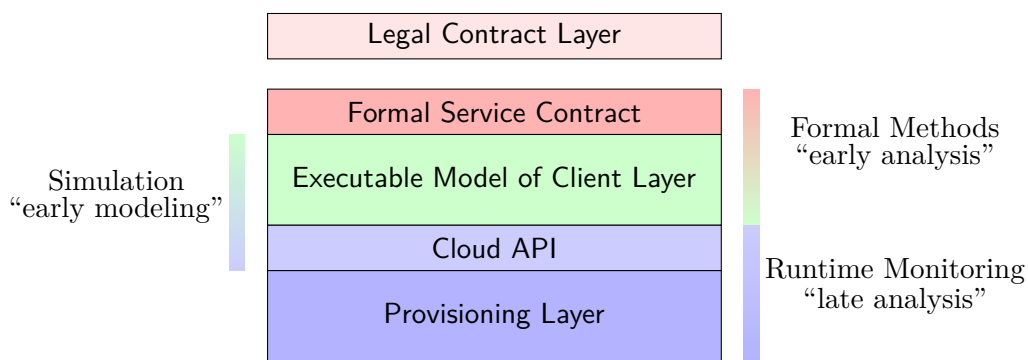


Figure 1.2: Making services resource-aware

The Envisage Framework. The Envisage framework will consist of an integrated suite of advanced methods and tools for engineering complex services that enables

- a design-by-contract methodology including service-level agreements,
- defining application-level services with resource requirements,
- modeling deployment scenarios reflecting elastic, virtualized architectures,
- a monitoring system assuring adaptability to failures and to renegotiations of service-level agreements,
- the systematic analysis of quality of service behavior of these models at early stages in software development.

The framework of Envisage is designed such that it allows the systematic combination of static and dynamic verification:

- formal, executable models
- a symbolic execution engine that can compute strongest post conditions in the form of symbolic execution states
- a contract-based specification framework
- support for runtime monitoring

1.2 Work Flows

It is important to notice that the methods and tool suite developed in **Envisage** do not impose a particular work flow for using the tools. To highlight this aspect of the **Envisage** outcomes, we will discuss two possible work flows in this deliverable.

- **Model-driven development of a new service.** Chapter 3 describes an iterative work flow for new services building on the **Envisage** framework;
- **Model-driven deployment of existing services.** Chapter 4 describes an iterative DevOps work flow to leverage the approach of **Envisage** for existing services.

Note that these two work flows are not mutually exclusive, as new services typically build on existing services.

1.3 Dissemination of the Proposed Methodology

This deliverable includes two papers describing the development methodology proposed by **Envisage**. The two papers, which are contributions to this deliverable, are:

- The position paper “Engineering Virtualized Services” [4] presents the vision of **Envisage** on software development at the start of the project. The paper is included as Appendix A to this deliverable.
- The paper “Designing Resource-Aware Cloud Applications” [11] argues why it is important to address deployment early in the development process in order to ensure a scalable design and reduce over-provisioning during service deployment, presenting our approach to service engineering. The paper is included as Appendix B to this deliverable.

Chapter 2

Background on Envisage Outcomes

2.1 Approach.

Figure 2.1 (taken from the DoW) depicts how cloud applications and service-level agreements are linked together based on formal models in **Envisage**. This chapter briefly summarizes the modeling artifacts and tools developed to address different parts of this tool chain.

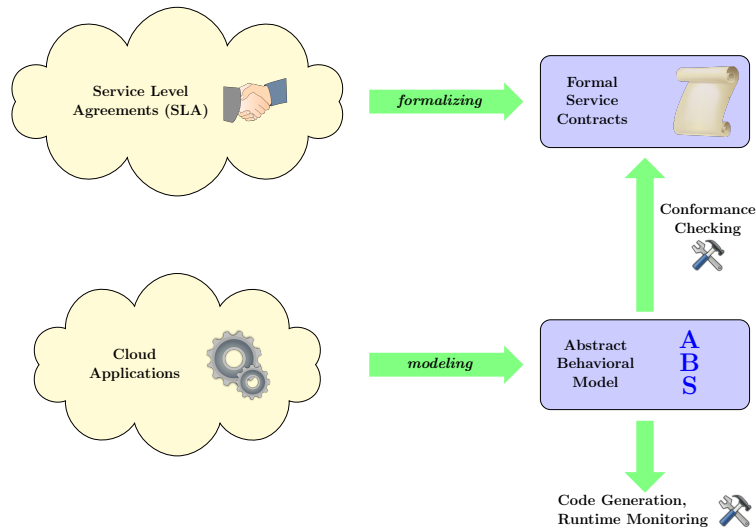


Figure 2.1: The Envisage tool chain.

2.2 Overview of the ABS Modeling Language.

ABS is a high-level executable object-oriented modeling language for distributed systems. It has a formal semantics to enable static analyses. The main features of ABS are:

- **Interfaces** similar to APIs can be extended to include information about functional and non-functional behaviors.
- **Concurrent objects**: Clean concurrency model and support for asynchronous communication
- **Deployment components**: Built-in modeling concepts for virtual resources
- **Separation of concerns** between the cost of execution and the capacity of the underlying platform

The formalization and use of ABS has been detailed in a number of papers (e.g., [5, 6, 8, 12]). Deliverable D1.3.1 provides guidelines in using ABS to model both static and dynamic deployment.

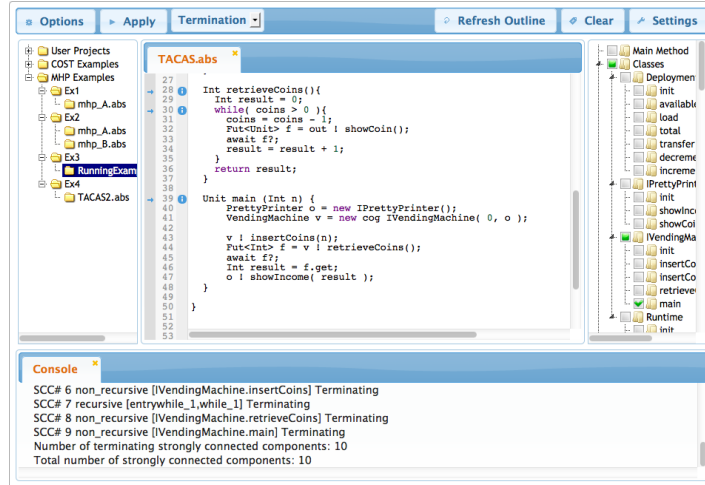


Figure 2.2: The web front-end of the ABS collaboratory.

2.3 Overview of the Analysis Tools Developed for ABS.

ABS is supported by a number of tool-based analysis methods, which are currently being developed in the Envisage project. We can classify these tools according to the arrows of Fig. 2.1:

conformance checking.

- **DF4ABS** supports the detection of deadlocks in the call graphs of ABS services [10] (see also Envisage Deliverable D2.1);
- **KeY-ABS** supports the verification of behavioral contracts for ABS services [7] (see also Envisage Deliverable D3.2);

Modeling & model exploration

- **Simulation tool** supports rapid prototyping and visualization of executable ABS models including static and dynamic resource management [12] (see also Envisage Deliverable D1.4.1);
- **MODDE** supports the computation of static deployments for ABS models of services [6] (see also Envisage Deliverable D1.3.1);
- **SACO** supports resource analysis of executable ABS models [1] (see also Envisage Deliverable D3.3.1);
- **TCG tool** to systematically test the ABS models [2, 3, 17] (deliverable D3.5 is due at M30);

Code generation & monitoring

- **Code generation tools** support the generation of code from executable ABS models into either Haskell or Java [13, 15, 16] (see also Envisage Deliverable D3.1);
- **Runtime monitoring framework** supports the generation of monitors for deployed services from ABS models [14] (see also Envisage Deliverable D2.3.1).

2.4 Overview of the Collaboratory.

The ABS Collaboratory (depicted in Figure 2.2) integrates the tools listed above in order to interact with ABS models and tools in a uniform way. The collaboratory can be installed locally using Vagrant, or accessed as a service via a web browser (this may limit the available tools to find a reasonable way of sandboxing the collaboratory). The Collaboratory is further detailed in Envisage Deliverable D5.2.1.

Chapter 3

A Model-Driven Development Process with ABS

A central point in the approach taken by *Envisage*, is that services need to be *designed for scalability*. When developing services using the ABS modeling language, modeling constructs to express deployment are available to the developer. These constructs are to a large extent orthogonal to the modeling of the functional behavior of a service, but ABS also allows the developer to experiment with customized load balancers and resource management strategies to ensure SLAs at service endpoints.

Figure 3.1 shows a schematic iterative workflow for the software development lifecycle. We will now detail the different phases of such an iterative workflow using the ABS tools. When developing new services, the developer may start from scratch (with “nothing” in terms of existing code) or building on top of legacy black-box services. In this chapter, we assume the developer starts from scratch, but we point out that ABS supports the inclusion of black-box external services during model-driven development in terms of very abstract models and a foreign language interface. hence, the workflow described in this chapter may integrate with the workflow of Chapter 4 to form “hybrid workflows” in various ways. There are no clear borders between the two workflows.

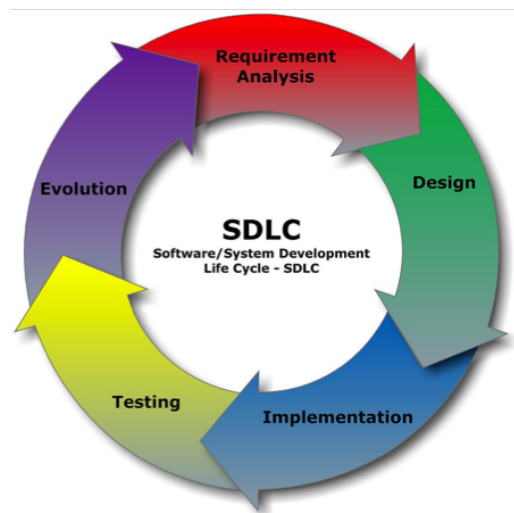


Figure 3.1: A schematic software development lifecycle emphasizing an iterative workflow.

3.1 Requirement Analysis

In addition to traditional functional requirements, this phase focusses on service levels and deployment costs. Since these requirements are internal to the service being developed, the non-functional requirements will typically relate external service delivery SLAs to the deployment costs of a service instance. Examples of such requirements could be that the service should be able to offer a portfolio of client SLAs associated with service endpoints (e.g, delivery times for x clients and scaling between thresholds of client traffic) with deployment profiles for each of these SLAs, as well as internal requirements including, as encountered in the ATB case study (Deliverable D4.2.2), the time needed before crawled data has led to updated indices on the mobile devices.

3.2 Design

The model-driven design phase consists of the development of ABS interfaces and classes, as well as making static and dynamic deployment decisions. In this phase, different parts of the service will coexist at different levels of abstraction, from detailed functional models of new and core functionality to abstract interface-level descriptions of libraries or underlying services expressed in terms of, e.g., best-case / worst-case response times or constant costs. The model interacts with cloud provisioning via the Cloud API, which is instantiated based on profiles of the resources available to the service. The developer may plan static deployments based on deployment constraints and devise dynamic resource management strategies based on interaction with the monitoring framework. In this phase, the developers may take advantage of many of the tools proposed in Envisage to help analyze their designs: rapid prototyping, test-case generation support to explore the model, static deployment analysis, deadlock analysis, cost analysis, as well as functional verification.

3.3 Implementation

An initial implementation can be obtained from the ABS model by means of the code generation tools offered in the ABS tool suite, in order to provide executable code from ABS models. This allows the developer to rapidly evaluate the integration with other services and the accuracy of the cloud resource profiles. Specific ABS libraries allow the developer to link their models directly into existing deployment frameworks such as Hadoop¹, its resource manager Yarn [9], or AWS² resource managers, but the code generation also supports customized resource management and scaling strategies, for example for hybrid cloud architectures.

3.4 Testing

Testing the deployment is done by means of the monitoring framework provided for ABS models. The tool suite allows local monitors to be deployed together with the generated code to provide dynamic analysis support for service delivery in the actual deployment. The monitoring framework gives the service a single entry point to the monitors associated with a service endpoint; e.g., the service capacity at a given endpoint can be controlled with respect to the agreed SLA.

3.5 Evolution

The work flow provides a feedback loop from (a) modeled deployment and (b) deployed software back to the requirement and design phases.

- The static analysis of the ABS model may provide feedback to the requirements; e.g., the resource requirements for certain service levels may be unrealistic, the service may not compose in a way which

¹<https://hadoop.apache.org>

²<https://aws.amazon.com>

supports the required scaling, or the deployment cost requirements cannot be met with the current resource profiles from the cloud provider. In **Envisage**, the feedback loop will concretely be made in terms of system traces violating the generated monitors, which can be rerun on the ABS models.

- The dynamic analysis of the deployed software may reveal errors in the assumptions of systems on which the service relies; e.g., the cloud provider may not deliver according to the resource profiles, the cost and timing information for existing services may be too imprecise or even incorrect, and there may be issues in the deployed service which were overlooked in the model.

Chapter 4

A DevOps Enabled Work Flow for ABS

The DevOps work flow emphasizes collaboration between software developers, operations personnel and quality assurance teams, recognizing interdependencies between software design, quality of service (QoS) and quality assurance. This is achieved by a recurring flow of rapid releases facilitated by automated configuration and continuous monitoring / testing. A schematic DevOps work flow is depicted in Figure 4.1.



Figure 4.1: A DevOps work flow connects development and operations in a continuous iterative process (illustration source: Gene Kim, HP, and PwC, 2013).

We now propose an ABS DevOps work flow which supports automated configuration by allowing the modeling at the *development* level and the deployment at the *operations* level go hand in hand; in particular, the *Envisage* approach allows configuration and deployment choices already at *the modeling level* for even very abstract models. This allows early exploration and analysis of different alternative deployments, thereby supporting the operations team to make informative choices, and supporting developers to quickly detect the possible need for further development iterations, in case the results are not satisfactory. We first describe how to set up an initial ABS model and then discuss each phase of the continuous iterative process in the figure separately.

4.1 Creating the ABS Model

To enable the DevOps work flow, we first need to create an ABS model. Our aim is to make this model as abstract as possible, omitting details of the behavior of the different parts wherever possible. The model

will consist of three parts:

- *Requirements.*
- *Deployment components and the cloud API.*
- *The executable ABS model of the services.*

Requirements. Requirements are specified in a high-level declarative language and can come from multiple stakeholders (e.g., domain knowledge, security, and business strategies). For instance, if two services are tightly coupled and communicate heavily, it may be beneficial to deploy them on the same virtual machine for technical (performance) reasons. On the other hand, requirements can also arise from security or business reasons: for instance, avoiding deploying services that operate on sensitive private customer data on virtual machines shared with services from other customers. Requirements may also relate to predefined customer level SLAs such as response times for a given number of users, etc.

Deployment components and the cloud API. The number of available virtual machines, their operational costs (typically determined by the underlying infrastructure provider) and the properties of the virtual machine can be specified in e.g., JSON format. In a cloud environment there is typically no a priori fixed number (or even an upper bound) of available machines and the only constraints on the virtual machines are the total cost of the entire environment (which is minimized) and whether a virtual machine is sufficiently powerful to offer the service(s). At the ABS level, virtual machines are represented through `DeploymentComponents` (see Envisage Deliverable D1.2.1). The properties of the virtual machines are given as parameters to the ABS Cloud API.

The executable ABS model of the services. The final input for the ABS model consists of the APIs associated with the services and the number of resources that a service consumes. We assume that we have available the APIs of the services we are modeling, and their interrelations (for example in the form of callgraphs). These resources can be specified in the form of annotations for the APIs of the different services. Dependencies between services (such as that a certain service can only be instantiated given an instance of another service) can also be expressed using the annotations. This information is sufficient to set up a coarse-grained executable model, based on stubs with timing, cost annotations, and auxiliary method calls.

To extract an ABS model the actual low-level implementation of the services is not strictly required, but only the APIs. If only the APIs are available, the user can choose to specify the resource consumption based on, for example, measurements taken from the in-production system. For existing or legacy systems where the implementation sources are available some further automation may be possible. If the system is implemented in Java (such as the Fredhopper Cloud Services), in principle the APIs can be extracted automatically due to the close correspondence between ABS and Java at the API level. The implementation can be made more precise in a stepwise manner by adding cost statements as a stand-in for parts of the implementation, and refining those iteratively with real implementations if a more detailed modeling is needed or desired to enable more precise analysis and thereby better deployments.

4.2 Automated Tests

A DevOps team takes advantage of test automation to ensure the quality of increments into the final service. Every commit apart from its build triggers a series of *automated* integrated tests. The integration tests can be performed in different levels of component integration, system integration and end-to-end testing. In addition, automated tests can verify the behavior of the system in terms of failure handling and load testing in a distributed test environment. In the context of the ABS models, testing takes the form of model-based simulations of given deployment scenarios for given client configurations combined with automatically generated test cases to explore the models.

Based on the formal semantics of ABS and the executable models, this phase includes the application of the formal analysis tools of **Envisage** (see Chapter 2) to the extent enabled by the level of detail in the model. The ABS models, which may be coarse-grained and with the possible presence of cost statements, can be used as an input to both static analyzers and dynamic techniques. For instance, resource analyzers like SACO (Envisage Deliverable D3.3.1) can statically automatically *infer* resource consumption based on the partial implementation. The cost statements are supported by SACO. In general, the presence of a more detailed implementation can improve accuracy and enables more powerful and fine-grained analyses, such as conformance checking by applying symbolic execution using KeY-ABS and deadlock detection using DF4ABS.

4.3 Automated Configuration

Roughly speaking, a resource configuration determines how services or objects are distributed over virtual machines. The goal is to find an optimal configuration of cloud resources, taking QoS requirements (for instance, derived from SLAs) and operational cost of the running virtual machines into account. Services can be deployed in two distinct phases:

1. Statically with MODDE (Deliverable D1.3.1), to find a suitable *initial* cloud resource configuration. This static configuration can also integrate dynamic resource managers expressed in ABS¹.
2. Dynamically, evolving the current configuration in a feedback loop (see Figure 4.1) through monitoring to react to changes in the environment, such as peaks in user demand or failing machines.

The configuration framework relies on input from the ABS model and the testing phase with model-based analysis. In particular, the suitability of a given configuration inherently depends on several factors:

- Are deployment requirements from each stakeholder taken into account?
- How many and what kind of virtual machines are available?
- What is the operational cost for each kind of virtual machine?
- How much resources are consumed by service instances?

Each of the above data forms an input to the framework (Figure 4.2) that synthesizes fully automatically an optimal resource configuration: a configuration that satisfies the requirements and in which the total number of used virtual machines has a minimal cost.

4.4 Continuous Monitoring and Feedback

The current resource consumption and operational cost of the running system can be dynamically retrieved through the Cloud API (Envisage Deliverable D1.3.1), by calling

- `DeploymentComponent.load`,
- `DeploymentComponent.total`, and
- `CloudProvider.getAccumulatedCost` respectively.

The returned information serves as an input to the monitoring framework (Envisage Deliverable D2.3.1). Operationally, the semantics of ABS models featuring cost statements is modeled faithfully in back-ends that support Real-time ABS. Currently Real-time ABS is supported by the Maude and Erlang backends

¹Although ABS supports dynamic resource management, MODDE does not currently support automatic configuration of dynamic resource management so dynamic managers must be selected in the testing phase and given to MODDE.

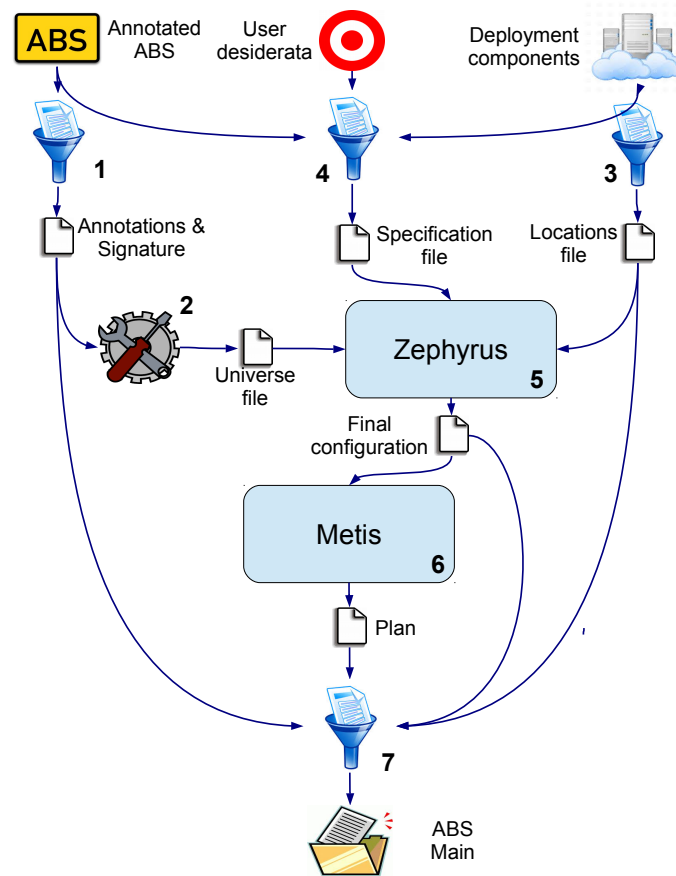


Figure 4.2: Execution flow to synthesize and analyze statically deployment configurations.

(the simulation backends, Envisage Deliverables D1.4.1 and D3.1). The monitoring framework is used to generate distributed monitors to be deployed together with the real, deployed services.

When the monitors detect a violation of the expected (non-functional) behavior, the monitoring framework returns an execution *trace* of the deployed service. We use this trace as a *driver* at the level of the ABS model to recreate the violation and refine and adjust the model in order to resolve the violation. Thus, the ABS model will be detailed and improved as a gradual process driven by feedback from the actual deployment of the services. An important aspect of this approach is that effort is only spent refining crucial parts of the ABS models, the other parts will remain at the initial, abstract level.

4.5 Collaborative Development

We see how the development of the ABS model co-exists with development of the code. As code moves to production, the ABS model evolves correspondingly. All the development team contributes and works on the same code repository. Every model and/or code change is presented as a commit in the code repository. Every commit passes through peer code reviews. Every commit after *successful* build and integrated tests is a candidate for a release, either at the modeling or code level. To support ABS for collaborative development, the ABS Collaboratory is being developed to integrate the Envisage technologies in one common frontend for the ABS part of the software development process. In addition, we have developed an Eclipse² plug-in for ABS.

²<https://eclipse.org>

4.6 Continuous Integration

Every development team ensures, at the beginning of the cycle of working a service or product, that necessary continuous integration (CI) tools are in place. Every commit triggers a new build in continuous integration system. The CI system also supports *supplementary* phases to further verify the quality of the commit. Examples include static code analysis checks, automatic deployment on a test environment, automatic release management. In addition, automated tests can be triggered via the CI. To try the CI, we have experimented with an ABS integration into Jenkins³ in order to run integrated tests when revising ABS models.

³<https://jenkins-ci.org>

Bibliography

- [1] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer-Verlag, 2014.
- [2] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Test Case Generation of Actor Systems. In *13th International Symposium on Automated Technology for Verification and Analysis, ATVA 2015. Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2015. To appear.
- [3] Elvira Albert, Nikolaos Bezirgiannis, Frank de Boer, and Enrique Martin-Martin. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. Submitted to FLOPS 2016.
- [4] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In M. Ali Babar and Marlon Dumas, editors, *2nd Nordic Symposium on Cloud Computing & Internet Technologies (NordiCloud’13)*, pages 59–63. ACM Press, 2013.
- [5] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatter, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, 2014.
- [6] Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. On the integration of automatic deployment into the ABS modeling language. In *Service Oriented and Cloud Computing - 4th European Conference, ESOC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, pages 40–64, 2015.
- [7] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International conference on Automated Deduction*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer-Verlag, 2015.
- [8] Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In Michael Butler et al, editor, *In Proceedings of the 17th International conference on Formal Engineering Methods (ICFEM 2015)*, volume 9407 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2015. to appear.
- [9] Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC’13)*, page 5. ACM, 2013.
- [10] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling*, 2014. To Appear.
- [11] Reiner Hähnle and Einar Broch Johnsen. Designing resource-aware cloud applications. *IEEE Computer*, 48(6):72–75, 2015.

- [12] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.
- [13] Behrooz Nobakht and Frank S. de Boer. Programming with actors in Java 8. In Tiziana Margaria and Bernhard Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*. Springer-Verlag, 2014.
- [14] Behrooz Nobakht, Stijn de Gouw, and Frank S. de Boer. Formal verification of service level agreements through distributed monitoring. In Schahram Dustdar, Frank Leymann Schahram, and Massimo Villari, editors, *Service Oriented and Cloud Computing – 4th European Conference, ESOC 2015, Taormina, Italy, September 15-17, 2015*, *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2015.
- [15] Vlad Serbanescu, Keyvan Azadbakht, Frank de Boer, Chetan Nagarajagowda, and Behrooz Nobakht. A design pattern for optimizations in data intensive applications using ABS and JAVA 8. *Concurrency and Computation: Practice and Experience*, 2015.
- [16] Vlad Serbanescu, Chetan Nagarajagowda, Keyvan Azadbakht, Frank de Boer, and Behrooz Nobakht. Towards type-based optimizations in distributed applications using ABS and JAVA 8. In *First International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, volume 8907 of *Lecture Notes in Computer Science*, pages 103–112. Springer-Verlag, 2014.
- [17] Peter Y. H. Wong, Richard Bubel, Frank S. de Boer, Miguel Gómez-Zamalloa, Stijn de Gouw, Reiner Hähnle, Karl Meinke, and Muddassar Azam Sindhu. Testing abstract behavioral specifications. *Journal on Software Tools for Technology Transfer*, 17(1):107–119, 2015.

Glossary

- Build** A process that compiles and builds the code repository for the software and generates an executable artifact. Examples include Jenkins, Bamboo, and TravisCI for the Java platform. 13, 15, 16
- Code Repository** A software configuration manager (SCM) that maintain the history of the source code and its configurations. 15
- Code Review** The process of reading team members code in the code repository and providing feedback in terms of comments or updated patches. 15
- Commit** A unique ID referring to a specific point in the history of the code repository. 13, 15, 16
- Conformance checking** Checking that an artefact conforms to a given property, in our case that an ABS model satisfies a given contract or requirement. 8
- Deployment** The process of shipping a released software package into an environment for execution. The process might include applying necessary configuration and eventually running the software package. 16
- DevOps** DevOps is a software development method that emphasizes the interdependence of software development, quality assurance (QA), and IT operations, and aims to help an organization rapidly produce software products and services and to improve operations performance. 6, 12
- Integrated test** A software test that is run to verify if the integration of different components is functionally correct after a code change in the system. 13, 15
- Release** The process that produces an increment to an executable artifact of the code repository. A release is commonly identified by a release version. The release process can be performed by the continuous integration system. 15, 16
- Sandboxing** Application sandboxing, also called application containerization, is an approach to software development that limits the environments in which certain code can execute. 8
- Static Code Analysis** A series of well-established static analysis techniques performed on the source code to extract code quality measures and anomalies. 16

Appendix A

Engineering Virtualized Services

Engineering Virtualized Services *

Elvira Albert

Complutense University of Madrid, Spain
elvira@fdi.ucm.es

Frank de Boer

CWI Amsterdam, The Netherlands
f.s.de.boer@cwi.nl

Reiner Hähnle

TU Darmstadt, Germany
haehnle@cs.tu-darmstadt.de

Einar Broch Johnsen
University of Oslo, Norway
einarj@ifi.uio.no

Cosimo Laneve
University of Bologna, Italy
laneve@cs.unibo.it

ABSTRACT

To foster the industrial adoption of virtualized services, it is necessary to address two important problems: (1) the efficient analysis, dynamic composition and deployment of services with qualitative and quantitative service levels and (2) the dynamic control of resources such as storage and processing capacities according to the internal policies of the services. The position supported in this paper is to overcome these problems by leveraging *service-level agreements* into software models and resource management into early phases of service design.

1. INTRODUCTION

Cloud computing is an execution environment with elastic resource provisioning, several stakeholders, and a metered service at multiple granularities for a specified level of quality of service (QoS) [10]. A host of cloud computing presents a number of services to client applications, including infrastructure and platform functionalities and software services for virtualizing the deployment of resources. This virtualization provides an elastic amount of resources to application-level services, thus making it possible to, for example, allocate a changing processing capacity to a service depending on demand. We say that application-level services are *virtualized* if they can adapt to the elasticity of cloud computing.

For virtualized services, resource provisioning is regulated by a legal contract between the service owner and the provider of the virtualized environment, called a *service-level agreement* (SLA). However, these legal texts are by their very nature not integrated in the software artifacts. Current modeling and analysis techniques make it extremely difficult for the software developer to realistically predict the resource requirements of the targeted service at an early design stage. Languages and tools for software development lack high-

level support to systematically analyze performance under varying resource assumptions and to express and compare different resource management policies. Variations in end-user scenarios, value-added services, and dynamic service composition further complicate the picture by extending the functionalities of an application-level service at the expense of potentially changing its cost profile.

In traditional engineering processes for services, both the deployment and the SLA regulating the deployment are add-ons to the software development process. The appropriate deployment and SLA compliance are determined a posteriori, *after* the design of the service's program logic. Virtualization allows deployment and resource provisioning to be *internalized* as part of the program's logic, enabling services to dynamically scale to accommodate client traffic.

For software development methods to be effective in the engineering of virtualized services, it is our position that

1. SLAs should be part of a *design by contract methodology* for virtualized service engineering and
2. virtualized resources should be managed by explicit language primitives since the early phases of service design.

These are key concepts that (i) enable the composition of virtualized services with respect to their quality and (ii) allow software developers to address the challenges posed by virtualization for the software-as-a-service abstraction already at early stages of development. Services for virtualized environments require descriptions of resource-dependent and resource-aware behaviors that are based on abstract yet detailed executable models. This helps to optimize the usage of runtime resources, as well as to decrease development costs and shorten time to market for service developers.

Our position calls for a model-based analysis of quantitative (non-functional) aspects of SLAs, rather than qualitative aspects of SLAs such as security. A major implication of our position is to enable a coherent tool-based analysis of *models of SLA-aware application-level services* in the context of different *deployment scenarios*. This means that models should

1. capture scalable services through their support for resource awareness and resource management, and
2. be analyzed by applying techniques that are based on scalable methods.

*This position paper is written in the context of the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (<http://www.envisage-project.eu>).

In the sequel, we detail our position on model-based analysis of SLAs in a design by contract methodology, and discuss its consequences for a research agenda in formal methods.

2. DESIGN BY CONTRACT

The term *design by contract* was coined by Bertrand Meyer referring to the contractual obligations that arise when invoking methods in the object-oriented programming language Eiffel [23]: only if a caller can ensure that certain behavioral conditions hold before the method is activated (the precondition), it is assured that the method results in a specified state when it completes (the postcondition). Design by contract enables software to be organized as encapsulated services with interfaces specifying the contract between the service and its clients. Clients can “program to interfaces”; they need not know how the service is implemented.

Our position necessitates a design by contract methodology for SLA-aware virtualized services, which is to be integrated in industrial software development processes. By investigating the contractual obligations that are present in a PaaS environment, specific abstractions can be identified that are suitable for collaboration between platform provider, service provider, and service client. We build on work on interface automata [9], typestates [29], user-defined (i.e., application-level) scheduling policies [3, 12], and process contracts [22] to define *service contracts*, a novel model of behavioral interfaces that specify the QoS and the resource usage in different deployment scenarios. The service contracts will be embedded in a modeling language for services and will be amenable to formal analysis.

Our position also implies the need to provide a range of tools for the analysis of models, based on formal techniques, that ensure conformance to application-level services. These tools enable the application of various analyses on the executable models already during the early design phase of the targeted service. This allows the developer to improve resource usage and QoS in deployment scenarios spanning from virtualized services for mobile users to resource provisioning in data centers. Regarding the analysis of the models, the design by contract approach ensures scalability of the analyses by compositionality: the encapsulated modules need only be checked with respect to their service contracts. To support a coherent and consistent suite of tools, both the modeling language and the service contracts will have a formal semantics. We finally remark that the description of different quantitative aspects in the service contracts drives both the *horizontal verification* of developed services, i.e. between service contracts themselves, and the *vertical verification*, i.e. between service contracts and the actual cost that can be reliably and automatically estimated for the models.

3. MODELING VIRTUALIZED SERVICES

General-purpose modeling languages exploit *abstraction* to reduce complexity [20]: descriptions primarily focus on the functional behavior and the logical composition of software. Industry-strength object-oriented programming and modeling languages, however, support different concurrency and interaction paradigms. The most prominent are multithreading and concurrent objects, using interaction mechanisms such as method calls, message passing, and shared resources. Researchers, including the authors of this paper, have de-

veloped a number of techniques to enable the compositional development of modular systems and the flexible reuse of components. However these techniques still overlook how a software’s deployment influences its behavior. This is highly problematic for modern software targeting, for example, cloud computing and reflective middleware, where virtualization technologies allow an application to *modify resources of its deployment scenario during execution* [4].

To fully exploit the potential of virtualization, it is important to make services both scalable and cost efficient by leveraging deployment decisions to the software design. A major challenge in software engineering today is to find a tradeoff between the two conflicting requirements of abstraction and deployment control in the application design phase. In fact, the introduction of low-level deployment in a high-level modeling language is potentially disruptive in software engineering, but it is unavoidable due to the new scenario that is delineated by cloud computing. It is worth noting that in software design, no general, systematic means exists today to model and analyze software in the context of a set of available virtualized resources, nor to analyze redistribution of virtualized resources in terms of load balancing or reflective operations. To the best of our knowledge, no current research directly addresses these challenges raised by virtualization, and in particular, the modeling of quantitative virtualized resources as data inside the software itself, which is a primary property of virtualized resources.

Our starting point is a separation of concerns between the application model, which requires resources, and the deployment scenario, which reflects the virtualized computing environment and elastically provides resources. This allows the developer to analyze the performance and scalability of a service for many different deployment scenarios already at the modeling level. For example, the model of an application may be analyzed with respect to deployments on virtual machines that may vary in a number of features: the amount of allocated computing or memory resources, the choice of application-level scheduling policies for client requests, and the distribution of computation over different virtual machines with fixed bandwidth constraints. Automated resource analysis [1] can be used to determine the most appropriate choice of SLA for the application, and to validate that the abstract system model complies with the SLA.

Models of virtualized systems in this context need to be SLA-aware: the modeling language will include primitives to express resource modeling and to support the virtualization of resources at an appropriate abstraction level. This way, the modeling language can express cloud computing software, such as SaaS business applications or PaaS abstractions, and feature an interface through which the application-level services can inspect and manipulate the virtualized resources of the platform. We see this interface in relation with standardization efforts in virtualization and cloud provisioning. The abstraction level of the modeling language also allows virtualized systems to be mapped to different deployment scenarios which describe the underlying virtualized architecture and to express dynamic load balancing policies depending on both the SLA and the current deployment of the service.

Executable models that describe precisely the control and data flow of the target service are a necessity for the analysis of the resource needs in different settings. Such executable models also allow code generation from the modeling language to different mainstream implementation languages, such as JAVA, SCALA, or ERLANG. Concrete starting points for such models are *abstract behavioral specification* languages, such as ABS [13]. The ABS language targets distributed systems based on object-oriented concepts, thus it may be easily used by software engineers, and service-level contracts can be naturally integrated into the object-oriented interfaces. These *service contracts* follow and extend the design by contract methodology, and include both behavioral interfaces and QoS descriptions.

As a proof of concept several models that include deployment scenarios with parametric resources have been created in ABS [14,15]. In these models application-level exchange of virtualized CPU resources is used to model and compare load balancing strategies between servers. Recently ABS has been applied to model dynamic resource management on the cloud [6,16]. These case studies show that our proposed formal approach compares favorably to custom simulation tools and that it scales to industrial problems. However, the proof of concept does not yet permit ABS models to be parametrized with resource policies in terms of SLAs, nor does it extend formal analysis to varying resource models and dynamic deployment.

4. FORMAL LANGUAGES FOR SLAS

The formalization of SLAs is a prerequisite for developing formal analysis methods that check whether a service conforms to an SLA. For this reason, a number of formal SLA specifications have been developed [2,18,30]. They all define SLAs in terms of XML schemata. The problem of all these notations is their lack of a formal semantics: they are all mark-up languages that rely on an implicit (hence inherently ambiguous) understanding of the various concepts represented. Another problem is that, for virtualized systems, SLAs will require continuous re-assessment, for the duration of the SLA, to cope with changing enterprise conditions. It is not clear how this continuous reassessment is addressed in the above proposals.

Specification languages for SLA are currently being integrated with semantic annotations, e.g., SAWSDL [19] for service descriptions and SWAPS [25] for WS-Agreement. Another relevant example is SLAng [21], an object-oriented language with a precise formal interpretation in terms of service infrastructure and behaviors. Similarly, Okika formalizes BPEL in a rewriting logic framework [24]. These efforts, however, have limited expressive power and the extension to elastic resources of virtualized systems has not been investigated. A different, more abstract SLA formalism is CC-pi [8], a combination of concurrent constraint programming and pi-calculus formalisms, which models computational processes for specifying and negotiating QoS requirements, and supports reasoning about resource allocation. CC-pi only checks for consistency and does not address issues as optimization of business values or contractual norms—topics that are addressed in detail in [27] and in RBSLA [26].

Today, client-level SLAs do not allow the service's potential resource usage to be determined or adapted when unforeseen changes to resources occur. This is because user-level SLAs are not explicitly related to actual performance metrics and configuration parameters of the services. As a result the recent EU FP7 project SLA@SOI [28], which is being continued in the Future Internet PPP project FIWARE [11], proposes an informal stepwise mapping between higher-level SLAs, such as those specified by clients, and lower-level SLAs and capabilities. This stepwise mapping is one of the prerequisites to support automated inference of resource usage from user-level SLAs.

Our position implies to push this line of research further and provide a modeling approach which *incorporates SLA requirements at the application-level* to ensure the QoS expectations of clients. This modeling approach will build on and consolidate the existing work to develop a practical, integrated SLA formalism for virtualized systems. It can be realized by investigating the contractual obligations that are present in, for example, a PaaS environment and provide specific abstractions that are suitable for the collaboration between platform provider, service provider, and client. The outcome of this consolidation effort includes a concrete SLA modeling language with a formal semantics and the formal description of (at least basic) enterprise level processes for SLA design and update. Another outcome is a contracts language that is embedded in an abstract behavioral modeling language such as ABS and thus amenable to formal analysis.

5. TOOLS FOR VIRTUALIZED SYSTEMS

Based on the formal semantics of an executable modeling language with integrated SLA, we envisage the development of a range of techniques for model-based analysis.

Monitoring and Service Contracts

Our position calls for techniques that address the difficulties of traditional monitoring tools [17], including fragmented visibility into the application stack, the lack of user-focused SLAs, and the absence of a budget perspective. The corresponding framework will provide a user-focused model with both a budget and cost perspective. Monitoring models must fill the gap between the negotiations with the client about SLAs, the service contract, and the deployment model.

Code Generation

Code generation for models of virtualized software should be instantiated to a deployment model. This will require developing new techniques, since models are rather high-level. However, automatic code generation will still be feasible, because models are executable and because of the constraints imposed by the deployment model. To prove correctness of the generated code, our position foresees the need for novel symbolic execution mechanisms that enable automated verification. Additionally, information about (asymptotic) resource consumption, computed by resource analysis of the high-level models, can be embedded into the profiles of the generated code. The correctness of such embedded information can be checked against the generated code to prove its validity, i.e., that the generated code preserves the resource consumption inferred from the model.

Resource Analysis

The cost analysis framework of virtualized services should be powerful enough to derive the deployment configuration and the interactions among services, and to automatically infer the overall cost from the cost of each service. The fact that this analysis will be developed at the level of the abstract models, which combine resource modeling with deployment modeling, allows these analyses to go beyond traditional cost models. For example, cost models for data size with bandwidth restrictions on communication can be developed for the underlying deployment scenarios. An important ambition (that stems from our overall position and that goes beyond the current existing technology) is to develop a resource analysis framework for determining whether certain resource usages are possible, given the service contracts of the component services.

Verification

We envisage an automatic deductive verification tool to ensure that a distributed, concurrent model respects a service contract. The properties stated in SLAs go beyond wellformedness of call sequences. For example, they involve limits imposed on storage space. Consequently, a first-order program calculus for the abstract modeling language is required. To achieve full automation, appropriate abstractions for service contracts need to be identified, following techniques suggested in [7], together with specialized proof search strategies and decision procedures. In case of a failed verification attempt, the developer should be supported by feedback on the kind of property that has been violated and under which condition. The developer also needs a concise rule book for modeling practices that help automation. In cases where automatic verification is still impossible, hybrid techniques can be considered, as discussed below.

Test Case Generation

Symbolic execution is the central part of most glassbox test case generation tools, which typically obtain the test cases from the branches of the symbolic execution tree. For virtualized services, the symbolic execution mechanisms should be integrated within a test case generation tool to produce test cases for the high-level models in a fully automatic way. The main challenge will be on handling distribution aspects of the services and the variety of deployment configurations within symbolic execution. Since information on resource management is explicitly available in our models it is possible to generate test cases that are aware of resource usage. A main research problem to be solved is *guidance* of the test case generator towards specific behaviors of the model by means of appropriate heuristics.

6. CONCLUSIONS

This position paper advocates a software engineering approach to virtualized services where (1) SLAs are part of a *design by contract methodology* and (2) virtualized resources are managed by explicit language primitives since the early phases of service design. This involves the extension of descriptions of virtualized services to encompass resource-dependent and resource-aware behaviors based on abstract yet detailed executable models. This new software engineering approach will render application-specific resource management policies to become fully integrated with the program logic of a service and analyzable already at an early

stage in the development of the service. This in turn leads to a better exploitation of runtime resources, as well as to lower development costs and shorter time to market for service developers.

The scale of the potential economical benefits inherent to our proposal can be illustrated by the well-known cost increase to fix defects in later development phases [5]. IBM Systems Sciences Institute estimates that a defect which costs one unit to fix in design, costs 15 units to fix in testing (system/acceptance) and 100 units or more to fix in production (see Figure 1, left), and this cost estimation does not even consider the *impact cost* due to, e.g., delayed time to market, lost revenue, lost customers, bad public relations, etc. Now, these ratios are for *static infrastructure*. Considering the high additional complexity of resource management for virtualized services, it is reasonable to expect even more significant differences; Figure 1 (right) conservatively suggests ratios for virtualized software in *dynamic infrastructures*. The modeling and analysis approach proposed in this paper aims at detecting deployment errors such as the impossibility to meet an SLA, already *in the design phase*. The associated savings potential clearly justifies any additional cost that might be incurred from formalisation of SLAs.

The research agenda proposed in this position paper forms the basis of a new EU FP7 project called ENVISAGE that includes (1) a behavioral specification language for describing resource-aware models and deployment choices; (2) a simulator with visualization facilities; and (3) tool support for automated resource analysis, validation of SLAs, code generation, and runtime monitoring of SLAs for deployed services. As argued above, such a methodology and associated tools will allow services to be delivered in a more effective, efficient, and reliable manner than today, accelerating the development cycle and lowering the operational costs for innovative networked services that make use of cloud computing.

7. REFERENCES

- [1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode, In *16th Eur. Symp. on Programming (ESOP)*, LNCS 4421, pages 157–172. Springer, 2007.
- [2] D. Battré, F. M. T. Brazier, K. P. Clark, M. A. Oey, A. Papaspyrou, O. Wäldrich, P. Wieder, and W. Ziegler. A proposal for WS-agreement negotiation. In *11th IEEE/ACM Intl. Conf. on Grid Computing*, pages 233–241. IEEE CS Press, 2010.
- [3] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
- [4] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 191–206. Springer, 1998.
- [5] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988.
- [6] F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte,

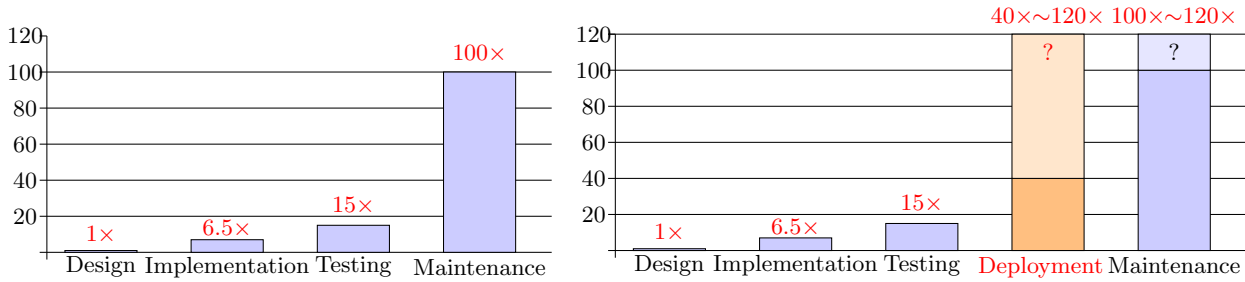


Figure 1: Relative costs to fix software defects for static infrastructure (left, source: IBM Systems Sciences Institute) and their extension to virtualized systems with dynamic infrastructure (right). The columns indicate the phase/stage of the software development at which the defect is found and fixed.

- and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. In *Eur. Conf. on Service-Oriented and Cloud Computing (ESOCC)*, LNCS 7592, pages 91–106. Springer, 2012.
- [7] R. Bubel, R. Hähnle, and B. Weiss. Abstract interpretation of symbolic execution with explicit state updates. In *6th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*. Springer, 2009.
- [8] M. G. Buscemi and U. Montanari. QoS negotiation in service composition. *J. Log. Algebr. Program.*, 80(1):13–24, 2011.
- [9] L. De Alfaro and T. A. Henzinger. Interface automata. In *8th Eur. Software Engineering Conf. & 9th ACM SIGSOFT Intl. Symp. on Foundations of software engineering, ESEC/FSE-9*, pages 109–120. ACM Press, 2001.
- [10] European Commission Expert Group Report. The future of cloud computing: Opportunities for European cloud computing beyond 2010, 2010.
- [11] FI-WARE. Web: www.fi-ppp.eu/projects/fi-ware.
- [12] M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.
- [13] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, LNCS 6957, pages 142–164. Springer, 2011.
- [14] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Intl. Conf. on Formal Engineering Methods (ICFEM)*, LNCS 6447, pages 646–661. Springer, 2010.
- [15] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Intl. Conf. on Formal Verification of Object-Oriented Software (FoVeOOS)*, LNCS 6528, pages 46–60. Springer, 2011.
- [16] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Intl. Conf. on Formal Engineering Methods (ICFEM)*, LNCS 7635, pages 71–86. Springer, 2012.
- [17] D. Jones. *The Definitive Guide to Monitoring the Data Center, Virtual Environments, and the Cloud*. Realtime publishers, 2010.
- [18] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:57–81, 2003.
- [19] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11:60–67, 2007.
- [20] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.
- [21] D. D. Lamanna, J. Skene, and W. Emmerich. SLAng: A language for defining service level agreements. *Future Trends of Distributed Computing Systems, IEEE Intl. Workshop*, page 100, 2003.
- [22] C. Laneve and L. Padovani. The must preorder revisited. In *18th Intl. Conf. on Concurrency Theory, LNCS 4703*, pages 212–225. Springer, 2007.
- [23] B. Meyer. Design by contract: The Eiffel method. In *TOOLS (26)*, page 446. IEEE CS Press, 1998.
- [24] J. C. Okika. *Analysis and Verification of Service Contracts*. PhD thesis, Aalborg University, 2010.
- [25] N. Oldham and K. Verma. Semantic WS-agreement partner selection. In *15th Intl. WWW Conf.*, pages 697–706. ACM Press, 2006.
- [26] A. Paschke. RBSLA a declarative rule-based service level agreement language based on RuleML. In *Intl. Conf. on Computational Intelligence for Modelling, Control and Automation and Intl. Conf. on Intelligent Agents, Web Technologies and Internet Commerce*, pages 308–314. IEEE CS Press, 2005.
- [27] J. P. Sauvé, F. Marques, A. Moura, M. C. Sampaio, J. Jornada, and E. Radziuk. SLA design from a business perspective. In *16th IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management (DSOM)*, LNCS 3775. Springer, 2005.
- [28] SLA@SOI. Web: <http://sla-at-soi.eu>.
- [29] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.
- [30] V. Tosic, B. Pagurek, and K. Patel. WSOL - a language for the formal specification of classes of service for web services. In *Intl. Conf. on Web Services (ICWS)*, pages 375–381. CSREA Press, 2003.

Appendix B

Designing Resource-Aware Cloud Applications

Resource-Aware Applications for the Cloud

Reiner Hähnle ¹ and Einar Broch Johnsen ²

¹ Technical University of Darmstadt, Germany haehnle@cs.tu-darmstadt.de

² University of Oslo, Norway enarj@ifi.uio.no

Making full usage of the potential of virtualized computation requires nothing less than to rethink the way in which we design and develop software.

Capitalizing on the Cloud

The planet's data storage and processing is about to move into the clouds. This has the potential to revolutionize how we will interact with computers in the future. A cloud consists of virtual computers that can only be accessed remotely. It is not a physical computer, you do not necessarily know where it is, but you can use it to store and process your data and you can access it at any time from your regular computer. If you still have an old-fashioned computer, that is. You might as well access your data or applications through your mobile device, for example while sitting on the bus.

Cloud-based data processing, or cloud computing, is more than just a convenient solution for individuals on the move. Although challenges remain concerning the privacy of data stored in the cloud, the cloud is already emerging as an economically interesting model for a start-up, an SME, or simply for a student who develops an app as a side project, due to an undeniable added value and compelling business drivers [1]. One such driver is *elasticity*: businesses pay for computing resources when they are needed, instead of provisioning in advance with huge upfront investments. New resources such as processing power or memory can be added to a virtual computer on the fly, or an additional virtual computer can be provided to the client application. Going beyond shared storage, the main potential in cloud computing lies in its scalable virtualized framework for data processing, which becomes a shared computing facility for your multiple devices. If a service uses cloud-based processing, its capacity can be automatically adjusted when new users arrive. Another driver is *agility*: new services can be deployed quickly and flexibly on the market at limited cost, without initial investments in hardware.

The EU believes¹ that cloud-based data processing will create 2.5 million new jobs and an annual value of 160 billion euros in Europe by 2020. Another study² predicts

¹ Digital Agenda for Europe, <http://ec.europa.eu/digital-agenda/en/european-cloud-computing-strategy>

14 million new jobs worldwide until 2015. However, reliability and control of resources are barriers to the industrial adoption of cloud computing today. To overcome these barriers and to regain control of the virtualized resources on the cloud, client services need to become resource-aware.

Challenges

Cloud computing is not merely a new technology for convenient and flexible data storage and implementation of services. Making full usage of the potential of virtualized computation requires nothing less than to rethink the way in which we design and develop software.

Empowering the Designer. The elasticity of software executed in the cloud means that designers have far reaching control over the resource parameters of the execution environment: the number and kind of processors, the amount of memory and storage capacity, and the bandwidth. These parameters can even be changed dynamically, at runtime. This means that the client of a cloud service not only can deploy and run software, but is also in full control of the trade-offs between the incurred cost and the delivered quality-of-service.

To realize these new possibilities, software in the cloud must be *designed for scalability*. Nowadays, software is often designed based on specific assumptions about deployment, including the size of data structures, the amount of RAM, and the number of processors. Rescaling requires extensive design changes, if scalability has not been taken into account from the start.

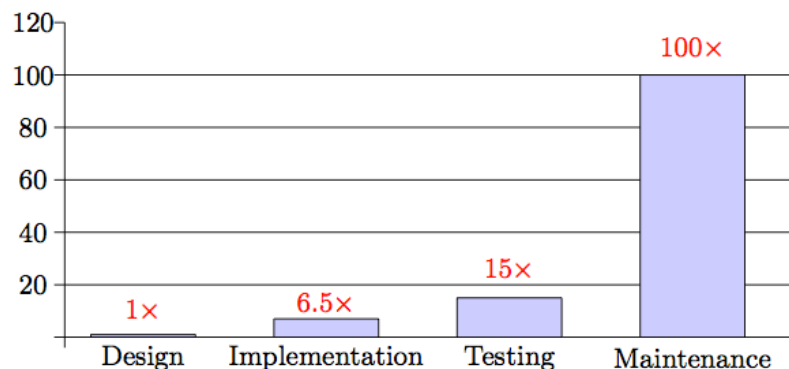


Fig. 1: Relative costs to fix software defects for static infrastructure (source: IBM Systems Sciences Institute). The columns indicate the phase of the software development at which the defect is found and fixed.

² IDC White Paper Cloud Computing's Role in Job Creation, <http://www.microsoft.com/en-us/news/features/2012/mar12/03-05cloudcomputingjobs.aspx>

Deployment Aspects at Design Time. The impact of cloud computing on software design, however, goes beyond scalability issues: traditionally, deployment is considered a late activity during software development. In the realm of cloud computing, this can be fatal. Consider the well-known cost increase for fixing defects during successive development phases [2]. IBM Systems Sciences Institute estimates that a defect which costs one unit to fix in design, costs 15 units to fix in testing (system/acceptance) and 100 units or more to fix in production (see Fig. 1). This cost estimation does not even consider the *impact cost* due to, for example, delayed time to market, lost revenue, lost customers, and bad public relations.

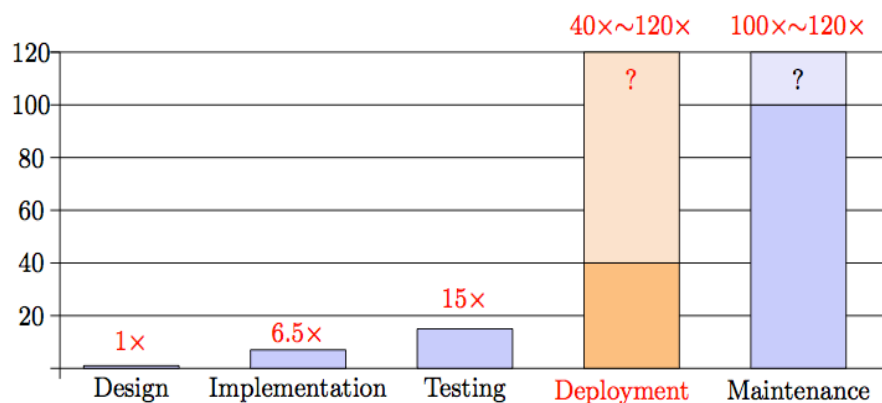


Fig. 2: Estimate of relative costs to fix software defects for virtualized systems with elasticity, from [3].

Now, these ratios are for *static* deployment. Considering the high additional complexity of resource management in virtualized environments, it is reasonable to expect even more significant differences; Fig. 2 conservatively suggests ratios for virtualized software in an *elastic* environment. This consideration makes it clear that it is essential to detect and fix deployment errors, for example, failure to meet a service level agreement (SLA), already *in the design phase*.

To make full usage of the opportunities of cloud computing, software development for the cloud demands a design methodology that (a) takes into account deployment modeling at *early* design stages and (b) permits the detection of *deployment errors* early and efficiently, helped by software tools, such as simulators, test generators, and static analyzers.

Controlling Deployment In The Design Phase

Our analysis exhibits a *software engineering challenge*: how can the validation of deployment decisions be pushed up to the modeling phase of the software development chain without convoluting the design with deployment details?

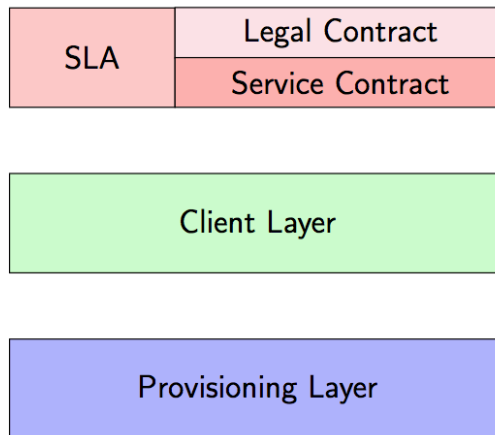


Fig. 3: Conceptual parts of a deployed cloud service.

When a service is developed today, the developers first design its functionality, then they determine which resources are needed for the service, and ultimately the provisioning of these resources is controlled through an SLA, see Fig. 3. The functionality is represented in the *client layer*. The *provisioning layer* makes resources available to the client layer and determines available memory, processing power, and bandwidth. The SLA is a legal document that clarifies what resources the provisioning layer should make available to the client service, what it cost, and states penalties for breach of agreement. A typical SLA covers two aspects: (i) a *legal contract* stating the mutual obligations and the consequences in case of a breach; (ii) the technical parameters and cost figures of the offered services, which we call the *service contract*.

How can the validation of deployment decisions be pushed up to the modeling phase of the software development chain without convoluting the design with deployment details?

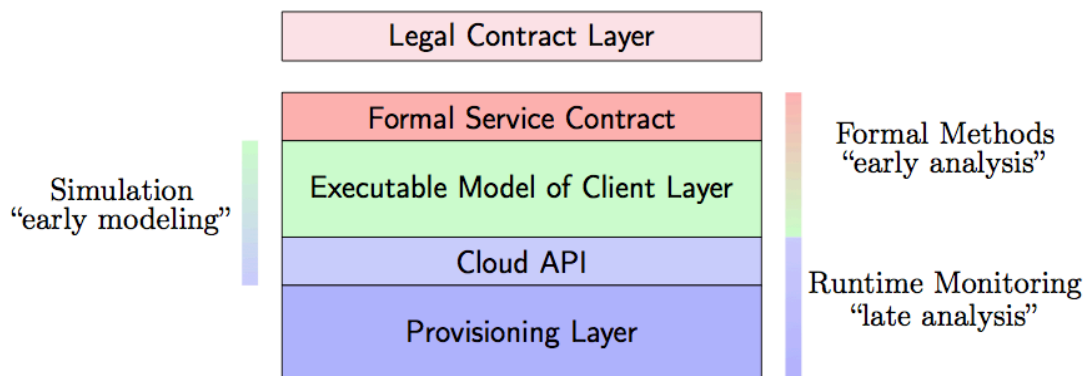


Fig. 4: Making services resource-aware.

So far, the different parts of a deployed cloud service live in separate worlds, but we need to connect them. In a first step the provisioning layer is made available to the client, so that the client can observe and modify resource parameters. We call this the *Cloud API*. This is *not* the same as the APIs that cloud environments provide to their clients now: our goal is to move deployment aspects into the *design phase*. We advocate that client behavior is represented by an *abstract behavioral model* of the client application, for example in an executable modeling language such as ABS [4]. Such a model can realistically be created during the design phase. The Cloud API is then an abstract interface to the provisioning layer, see Fig. 4. Such “early modeling” of client behavior makes it possible to simulate different client-side provisioning schemes and observe their impact on cost and performance.

To connect SLAs with the client layer, the key observation is that the service contract aspects of an SLA can be given a formal semantics. This enables formal analysis of client behavior with respect to the SLA at *design time*. Possible analyses include resource consumption, performance, test case generation, and even functional verification [3]. For modeling languages such as ABS this is highly automated [5]. “Early analysis” makes assumptions about the Cloud API explicit and enables the generation of monitors in the provisioning layer. Runtime monitors then provide “late analysis”.

Opportunities

Making deployment decisions at design-time shifts control from the provisioning layer to the client layer. The client service becomes resource-aware. This provides a number of attractive opportunities.

Fine-grained provisioning. Business models for resource provisioning on the cloud are becoming similarly fine-grained as those we know from other industry sectors such as telephony or electricity. It is becoming increasingly complex to decide which model to select for your software. Design-time analysis and comparison of deployment decisions allow an application to be deployed according to the optimal payment model for the expected end-users. Cloud customers can take advantage of fine-grained provisioning schemes such as spot price.

Tighter provisioning. Better profiles of the resource needs of the client layer help cloud providers to avoid over-provisioning to meet their SLAs. Better usage of the resources means more clients can be served with the same amount of hardware in the data center, without violating SLAs and incurring penalties.

Application-specific resource control. Design-time analysis of scalability enables the client layer to make better use of the elasticity offered by the cloud, to know beforehand at what load thresholds it is necessary to scale up the deployment to avoid breaking SLAs and disappointing the expectations of the end-users.

Application-controlled elasticity. Going one step further, we foresee autonomous, resource-aware services that run their own deployment strategy. Such a service will monitor the load on its virtual machine instances as well as the end-user traffic, and make its own decisions about the trade-offs between the delivered quality of service and the incurred cost. The service interacts with the provisioning layer through an API to dynamically scale up or down. The service may even request or bid for virtual machine instances with given profiles on the virtual resource market place of the future!

Summary

We argued that the efficiency and performance of cloud-based services are boosted by moving deployment decisions up the development chain. Resource-aware services give the client better control of resource usage, to meet SLAs at lower cost. We identify formal methods, executable models, and deployment modeling as the ingredients that can make this vision happen. A concrete realization of our ideas is currently being implemented as part of the EU FP7 project Envisage: Engineering Virtualized Services (<http://www.envisage-project.eu>).

Acknowledgments. This work has been partially supported by EU project FP7-610582 Envisage: Engineering Virtualized Services.

References

- [1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Sys.*, 25(6):599–616, 2009.
- [2] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. SW Eng.*, 14(10):1462–1477, 1988.
- [3] E. Albert, F. de Boer, R. Hähnle, E. B. Johnsen, and C. Laneve. Engineering virtualized services. In M. A. Babar and M. Dumas, editors, *2nd Nordic Symp. Cloud Computing & Internet Technologies*, pages 59–63. ACM, 2013.
- [4] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects*, volume 6957 of LNCS, pages 142–164. Springer, 2011.
- [5] E. Albert, F. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. *J. of Service-Oriented Computing and Applications*, 2013. Springer Online First, DOI 10.1007/s11761-013-0148-0.