Chapter 9
Models@Runtime for Continuous Design and Deployment

Nicolas Ferry and Arnor Solberg

9.1 Introduction

Nowadays, software systems are leveraging upon an aggregation of dedicated infrastructures and platforms, which leads to the design of large scale, distributed, and dynamic systems. The need to evolve and update such systems after delivery is often inevitable, for example, due to changes in the requirements, maintenance, or needs for advancing the quality of services such as scalability and performances. The demands to evolve and update the systems typically increase with Cloud-based systems, since the Cloud enable to dynamically adjust and evolve the platforms and infrastructures, while previously these were very much rigid and more or less fixed. This implies on the one hand more opportunities and flexibility to better evolve and adjust the systems to various needs and requirements, on the other hand the complexity of designing, delivering, managing and maintaining such systems challenges current software engineering techniques.

As stated in [1], in order to reduce delivery time and fostering continuous evolution of these systems, there is a need to close the gap between development and operation activities. However, developers and operators are often working in separate teams with specific roles, and thus, prefer to use the specific languages they feel comfortable with. This hinders the knowledge sharing between these teams, thereby, on the one hand making it difficult for designers to obtain and understand feedback on the status of the operated system that could be useful to evolve it, and on the other hand making it difficult for operators to analyse and comment on the impact of proposed or implemented design changes. As promoted by the DevOps movement [2]. This issue can be better handled by facilitating collaboration between developers and

N. Ferry · A. Solberg (✉)
Stiftelsen SINTEF, Postboks 4760 Sluppen, 7465 Trondheim, Norway
e-mail: Arnor.Solberg@sintef.no

N. Ferry
e-mail: Nicolas.Ferry@sintef.no

E. Di Nitto et al. (eds.), *Model-Driven Development and Operation of Multi-Cloud Applications*, PoliMI SpringerBriefs,
DOI 10.1007/978-3-319-46031-4_9

operators for example through aligning concepts and languages used in development and operation, and supporting them with automated tools that help reducing the gap and improving the flexibility and efficiency of the delivery life-cycle (e.g., resource provisioning and deployment).

In particular, continuous integration [3] tools play a key role, for example, through the significant increase of the frequency of integration it ensures immediate feedback to developers. Continuous integration also enable frequent releases, more control in terms of predictability (as opposed to integration surprises in less frequent and more heavy integration cycles) as well as productivity and communication. Continuous deployment can be seen as a part of the continuous integration practice and is defined as: "*Continuous deployment is the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users*" [3].

In the context of Cloud applications and multi-Cloud applications [4] (i.e., applications that can be deployed across multiple Cloud infrastructures and platforms), designers and operators typically seek to exploit the peculiarities of the many existing Cloud solutions and to optimise performance, availability, and cost. In such context, there is a pressing need for tools supporting automated and continuous deployment to reduce time-to-market but also to facilitate testing and validation of the design choices. However, current approaches are not sufficient to properly manage the complexity of the development and administration of multi-Cloud systems [5].

In this chapter we present the mechanism and tooling within the MODAClouds approach to reduce the gap between developers and operators by supporting continuous deployment of multi-Cloud applications. In order to reduce the gap between developers and operators we apply the same concepts and language for deployment and resource provisioning at development time and at operation time (the CLOUDMLpresented in Chap. 3). To automate the continous deployment and resource provisioning we have developed a deployment and resource provisioning engine based on the principles of the Models@Runtime approach [6]. This engine is responsible for enacting the continuous deployment of multi-Cloud applications as well as the dynamic adaptation of their deployment and resource provisioning including operations such as scaling out and bursting of parts of an application. The engine "speaks" the language of CLOUDML, thus, it provides the same concepts and abstractions for the operators as applied by the developers.

The remainder of the paper is organised as follows. Section 9.2 presents our model-based approach. Section 9.3 provides an overview of the MODAClouds Models@Runtime engine. Sections 9.3.1 and 9.3.2 details how the engine can be used to continuously adapt the deployment of an application in a declarative and imperative way, respectively. Section 9.3.3 presents the mechanism to monitor the status of the running system. Section 9.3.4 details the mechanisms enabling remote interaction with the engine. Finally, Sect. 9.4 presents some related work and Sect. 9.5 draws some conclusions.

9.2 The Models@Runtime Approach

Model-Driven Engineering (MDE) techniques have shown to be effective in supporting design activities [7]. MDE is a branch of software engineering which aims at improving the productivity, quality and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. Models and modelling languages, as the main artefacts of the development process, enable developers to work at a higher level of abstraction rather than at the level of implementation details. However, as stated in [6], applying the classical MDE approach for software evolution would be impractical. Indeed, this would typically result in generating the new solution, stopping the running system before replacing it by the new one, this in contrast with common expectations for Cloud services to have more or less 100 % up-time. In order to address this issue, the Models@Runtime approach has emerged.

Models@Runtime [6, 8] is an architectural pattern for dynamic adaptive systems that leverage models as executable artefacts supporting the execution of the system. This way, Models@Runtime promotes the DevOps method, by providing a unique model-based representation of the applications for both design- and run-time activities (i.e., for developers and operators). As depicted in Fig. 9.1, Models@Runtime provides an abstract representation of the underlying running system, which facilitates reasoning, simulation, and enactment of adaptation actions. A change in the running system is automatically reflected in the model of the current system. Similarly, a modification to this model is enacted on the running system on demand. This causal connection enables the continuous evolution of the system with no strict boundaries between design-time and run-time activities.

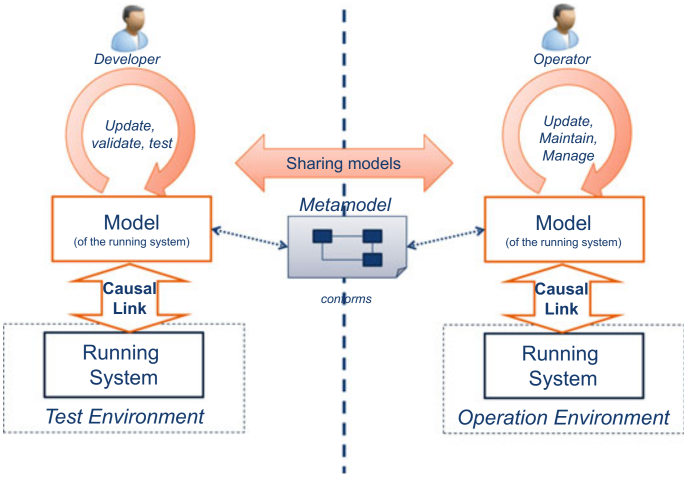

Fig. 9.1 Continuous deployment using Models@Runtime

Exploiting Models@Runtime for the continuous deployment of Cloud-based applications would thus result in the process depicted in Fig. 9.1. A developer team can specify a model of the deployment of its application (typically exploiting a domain-specific language such as CLOUDML) and thus automatically enact this deployment into a test environment. The team can therefore benefit from this test environment to tune its development and redeploy it automatically. Any change made to the deployment model will be enacted on demand on the running system whilst its status will be reflected in the model providing useful feedback. Once the new release is validated, it can be provided together with the associated deployment model to the operation team. The latter can in turn exploit the model to deploy the new release in a production environment. The operators can thus tune this model to maintain and manage the running system. Because the models shared by the developers and operators conform to the same metamodel, at any time they can share and exchange information.

9.3 The MODAClouds Models@Runtime Engine

The MODAClouds Models@Runtime environment relies on the Cloud Modelling Language [9] (CLOUDML) in order to provide a deployment model causally connected to the running system. As a result, the Models@Runtime maintains deployment models at two levels of abstraction: Cloud provider-independent models (CPIM) and Cloud provider-specific models (CPSM) as advocated by MODA-CloudML. On the one hand, any modification to the CPIM will be reflected in the CPSM and, in turn, propagated on-demand onto the running system. On the other hand, any change in the running system will be reflected in the CPSM, which, in turn, can be assessed with respect to the CPIM. This way, by exploiting the MODA-CloudML deployment model, the Models@Runtime environment seamlessly bridges the gap between the runtime and design-time activities. Figure 9.2 shows the CPSM of the Constellation case study (see Chap. 13) defined using the MODAClouds IDE and managed by the Models@Runtime engine.

Figure 9.3 depicts the architecture of the MODAClouds Models@Runtime engine. A reasoning system can read the current CPSM (step 1), which describes the actual running system, and produces a target CPSM (step 2). Then, the runtime environment calculates the difference between the current CPSM and the target CPSM (step 3). Finally, the adaptation engine enacts the adaptation modifying only the parts of the system necessary to account for the difference, and the target CPSM becomes the current CPSM (step 4). For each modification of the running system, the synchronization engine propagate notifications describing the change to third party entities.

Once the application is deployed, the Models@Runtime engine interacts with the Cloud providers API in order to observe the status of the Cloud services used. This mechanism is based on a pulling approach for which the frequency of the requests to the providers API can be parameterized.

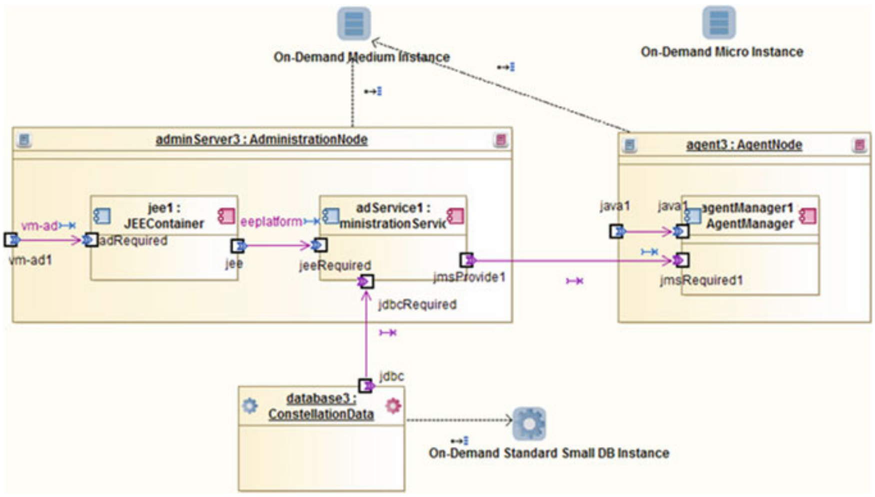
On-Demand Medium Instance
On-Demand Micro Instance
adminServer3 : AdministrationNode
agent3 : AgentNode
jee1 : JEEContainer
adService1 : ministrationServic
agentManager1 AgentManager
vm-ad
vm-ad1
adRequired
eeplatform
jee
jeeRequired
jmsProvide1
jdbcRequired
java1
jmsRequired1
jdbc
database3 : ConstellationData
On-Demand Standard Small DB Instance

Fig. 9.2 CPSM of the Constellation case study

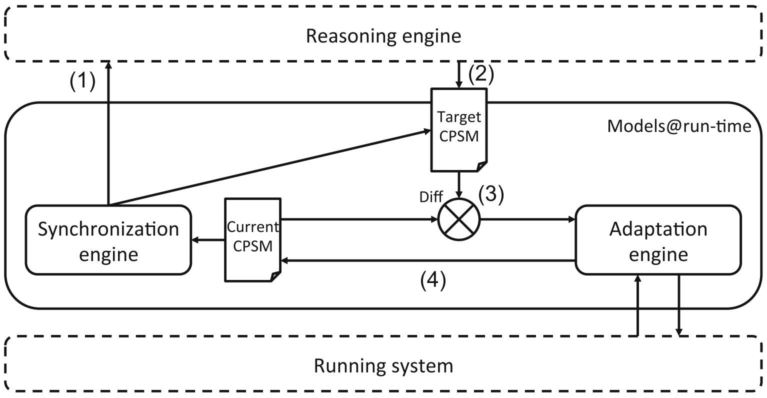

Fig. 9.3 The CloudML Models@Runtime architecture

Using the Models@Runtime engine, the deployment of an application can be adapted in both imperative and declarative ways. The imperative approach requires the explicit definition through a set of predefined instructions of how to reach the desired deployment. In contrast, the declarative approach requires the specification of the desired deployment and then the plan on how to reach that deployment is derived automatically. Both approaches result in a target CPSM that is consumed by a comparison engine, which computes the difference between the target model and the model of the running system. The result of this process is thus exploited to manipulate and adapt only the parts of the system necessary to account for the difference. In the following subsections we detail first the comparison engine and then the main adaptation commands.

9.3.1 The Comparison Engine

The inputs to the `Comparison engine` (also called `Diff`) are the current and target deployment models. The output is a list of actions representing the required changes to transform the current model into the target model. The types of potential actions are listed in Table 9.1 and result in: (i) modification of the deployment and resource provisioning topology, (ii) modifications of the components' properties, or (iii) modifications of their status on the basis of their life-cycle. In particular, the status of an external component (i.e., representing a VM or a PaaS solution) can be: `running`, `stopped` or in `error`, whilst the status of an internal component (i.e., representing the software to be deployed on an enternal component) can be: `uninstalled`, `installed`, `configured`, `running` or in `error`.

The comparison engine processes the entities composing the deployment models in the following order: `external components`, `internal components`, `execution binding`, to `relationships`, on the basis of the logical dependencies between these concepts. In this way, all the components required by another component are deployed first. For each of these concepts, the engine compare the two sets of instances from the current and target models. This comparison is achieved

Table 9.1 Types of output actions generated by the `Comparison engine`

Action	Parameter	Effect
addExternalComponent	ExternalComponent	Provision a new virtual machine or prepare a PaaS service
removeExternalComponent	ExternalComponent	Terminate a virtual machine or stop a PaaS service
addInternalComponent	InternalComponent	Deploy the internal component on the target virtual machine
removeInternalComponent	InternalComponent	Remove the internal component instance from its current host
addCommunication	Communication	Configure the endpoints of the communication
removeCommunication	Communication	Disconnect the endpoints of the communication
addHosting	Hosting	Configure the endpoints of the hosting
removeHosting	Hosting	Disconnect the endpoints of the hosting
setStatus	Status	Change the status of a component
setProperty	Property	Change a property of a component

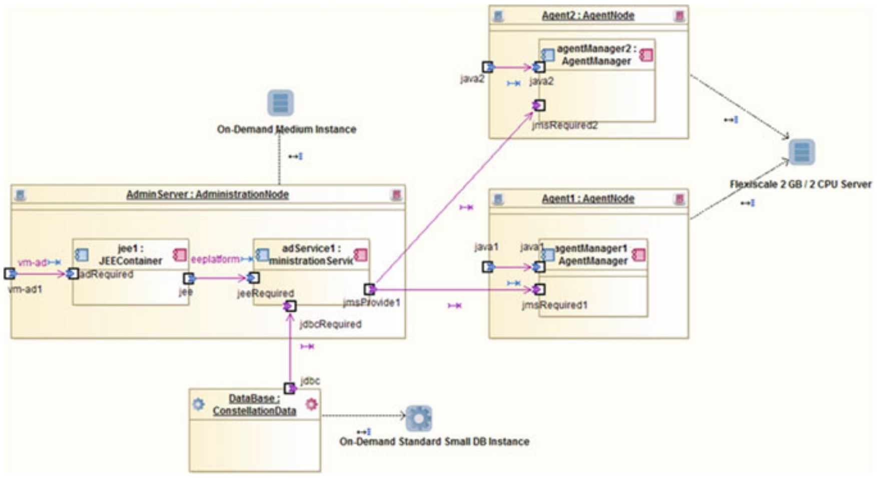
Agent2 : AgentNode
agentManager2 : AgentManager
java2
java2
jmsRequired2
On-Demand Medium Instance
Flexiscale 2 GB / 2 CPU Server
AdminServer : AdministrationNode
Agent1 : AgentNode
jee1 : JEEContainer
adService1 : AdministrationService
vm-ad
eeplatform
adRequired
vm-ad1
jee
jeeRequired
jmsProvide1
java1
java1
agentManager1 : AgentManager
jmsRequired1
jdbcRequired
jdbc
DataBase : ConstellationData
On-Demand Standard Small DB Instance

Fig. 9.4 An example of target CPSM of the Constellation case study

on the matching of both the properties of the instances and their types as well as on the basis of their dependencies (e.g., if the host of a component has changed the component might be redeployed). For each unmatched instance from the current model a `remove` action with the instance as argument is created. Similarly, for each unmatched instance from the target model an `add` action with the instance as argument is generated.

As an example, the comparison between the models depicted in Figs. 9.2 and 9.4 results in the following modifications in the deployment of the Constellation server: a new VM is provisioned on Flexiscale, Agent 1 is migrated from the on-demand medium instance to the new VM, and finally a new Agent is also installed on the same VM.

We always give a higher priority to the target model, which for example means that any virtual machine instance in the target model that does exist in the current model will be regarded as one that need to be created. Conversely, any virtual machine in the current model that does not exist in the target model will be removed. Coping with changes that happens during reasoning could be handled in various ways, for instance as part of a third step of the adaptation process (model checking). Currently, the Models@Runtime engine does not handle changes that might occur during the time of reasoning.

9.3.2 Adaptation Commands

As stated before, the deployment of an application can be dynamically adapted by exploiting the set of commands exposed by the engine. In particular, within the MODAClouds runtime environment, the Models@Runtime engine is responsible for enacting adaptation actions such as the scaling and bursting of an application.

These actions can be achieved by directly providing a deployment model to the Models@Runtime engine. For instance, the simplest way to perform a bursting at the IaaS level consists in updating the model of the running system by either updating the provider associated to the type of the VM instance or by simply changing the type of a VM instance with one associated to the desired provider. This approach allows fine grained tuning of the deployment of an application to the needs of new contexts or requirements, however, it can be a complex task for a third party to be responsible for evolving to the new deployment model.

Therefore, the Models@Runtime engine also provides high level commands that avoid direct manipulation of the models. In particular, the `scale` command enable scaling out a VM in the same Cloud and the `burst` command enable scaling out a VM in another Cloud. Currently, in both these cases the first task of the engine consists in modifying the current deployment model as follow:

1. Create a new instance of VM with unique name and port names of the same type as the VM to be scaled. In case of bursting, the provider associated to the new instance is the one specified in the bursting command.
2. For each internal component instance running on the VM to be scaled, create an instance of the same type and add an execution binding between each of them and the newly created VM. All new instances are created with unique names and port names.
3. Identify all the relationship instances involving the internal component running on the VM to be scaled and for each of them, create an instance of the same type with unique names. The endpoints of these new relationship instances are: the newly created internal component instance and the same component as the one involved in the original relationship.

Once the deployment model is updated, the engine acts differently depending of the type of command. In case of bursting to a new provider, the engine simply exploit the Models@Runtime comparison mechanism and trigger a classical deployment, whilst in the case of scaling within the same Cloud it operates as follows:

1. If not existing, create an image of the VM to be scaled.
2. Provision a VM using this image.
3. Reconfigure all components on the basis of the newly created relationship.
4. Restart the new components.

In case a set of VM instances cannot be further scaled (e.g., in case there are no more resources available on a private Cloud), the Models@Runtime engine acts as follows: The target model generated by the scale out command is considered as the current model of the system and the status of the newly created VM is set to `error` whilst the status of its hosted internal components is set to `unrecognized`.

In order to reduce the time needed to scale a VM, another provided feature is to provision VMs in advance with all the required software component deployed on it, and thus making them ready to be started or stopped on demand. In order to support such an approach, the Models@Runtime engine offers commands to start and stop components. These commands can be applied to both external and internal components. In the case of external components, this is achieved by exploiting the various Cloud provider APIs, whilst in the case of internal components it consists in calling the start and stop commands of the resources associated to the component. In both cases, the components have to be provisioned and installed upfront.

9.3.3 State Tracking

The Models@Runtime engine allows tracking the status of a deployment or adaptation as well as the status of Cloud resources once a multi-Cloud application is deployed. In order to track the state of Cloud resources, a simple monitoring agent is started in a parallel thread. Modules (one for each provider) can then be attached to the agent which are then responsible for interacting with the providers API in order to monitor the status of the Cloud resources being used. The frequency at which these status checks are performed can be configured manually or programmatically. Once performed and in case the status of a Cloud resource has changed, the agent exploits the Models@Runtime synchronization mechanism in order to reflect this change into the CPSM of the running system. As a result, all the registered clients of the Models@Runtime engine are notified of the update. Similarly, the status of the internal component is changed during the deployment process depending on the result of each deployment command.

The Models@Runtime engine is also synchronized with the MODAClouds monitoring platform (see Chap. 5) so that it can subscribe to receive some of the metrics collected by the monitoring platform.

In addition, this synchronization enable the co-evolution of the monitoring platform with the Cloud-application (e.g., when a service bursts from one provider to another, the monitoring activity has to be adapted accordingly). By synchronizing the Models@Runtime engine and the monitoring platform, the latter can dynamically and automatically be adapted to best fit with the actual deployment of the application.

In case the deployment of an application is adapted, the Models@Runtime engine, can communicate the changes to the monitoring platform and update the deployment of the data collectors. The monitoring platform can in turn adapt its own configuration accordingly, exploiting the Monitoring Manager which is the main coordinator of the monitoring activity. It manages and configures all the monitoring components including the model used by the Data Collectors (DCs) so that the retrieving of data can be adapted accordingly.

The deployment or un-deployment of Data Collectors can be done for example, to free resources, to replace a Data Collector with a new one that may offer slightly some different features, or when a monitored component is migrated. In addition, when the deployment of the running system is modified (e.g., bursting or migration from one provider to another), the monitoring activity will restart on the new machine using the same settings and rules used on the old one. Since the Models@Runtime engine can manage multi-Cloud applications and because the DCs are provider-agnostic, the migration can be performed from one provider to another.

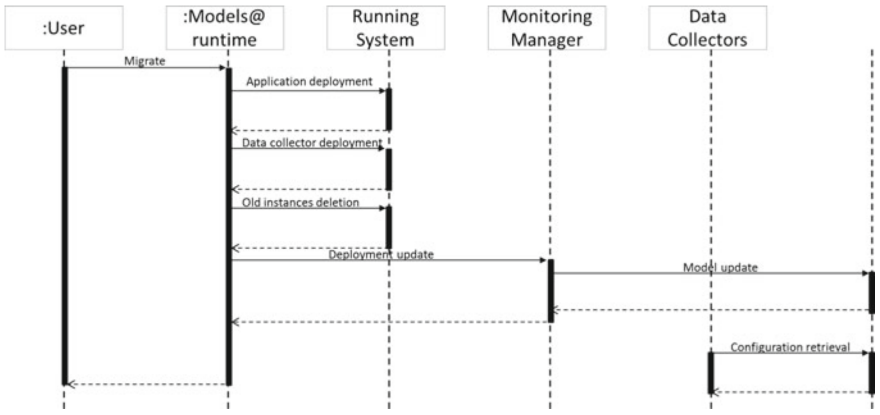

Fig. 9.5 Adaptation of the monitoring platform during the bursting process

Figure 9.5 details the interactions between the reasoning engine, the monitoring platform and the Models@Runtime engine during the migration of an application.

First, the Models@Runtime engine instantiates a new machine and deploys the application on it. Then it deploys the Data Collectors on the VM and finally removes the old instantiation of the application. At this stage, the Models@Runtime engine notifies to the Monitoring Manager the changes in the deployment (e.g., status of the new machine, address of the Data Collector), and the Monitoring Manager uses these information to autonomously update the KB from which the Data Collector retrieve its own configuration.

The communication from the model@runtime engine to the monitoring platform is performed through the REST APIs offered by the Monitoring Manager which is the main coordinator of the monitoring activity.

9.3.4 Interaction with the Models@Runtime Engine

The Models@Runtime environment also provides synchronisation mechanisms for remote third-party entities (e.g., such as the MODAClouds reasoning engines) to adapt the system. This synchronisation is implemented by the propagation of changes in both directions, namely `notification` and `command`. A notification allows the Models@Runtime engine to propagate its change to third-parties, whilst a command enables modifications on the current CPSM. This mechanism is exploited by various MODAClouds runtime components such as the MODAClouds reasoning engine to be informed of the changes occurring in the deployment of the running system and then adapt it accordingly. Because the two models used by two players can be isolated from each other and might not be aware of the whole model state, only the sequence of modifications is propagated, without carrying the start state of each change. Therefore, either notification or command is a sequence of modifications.

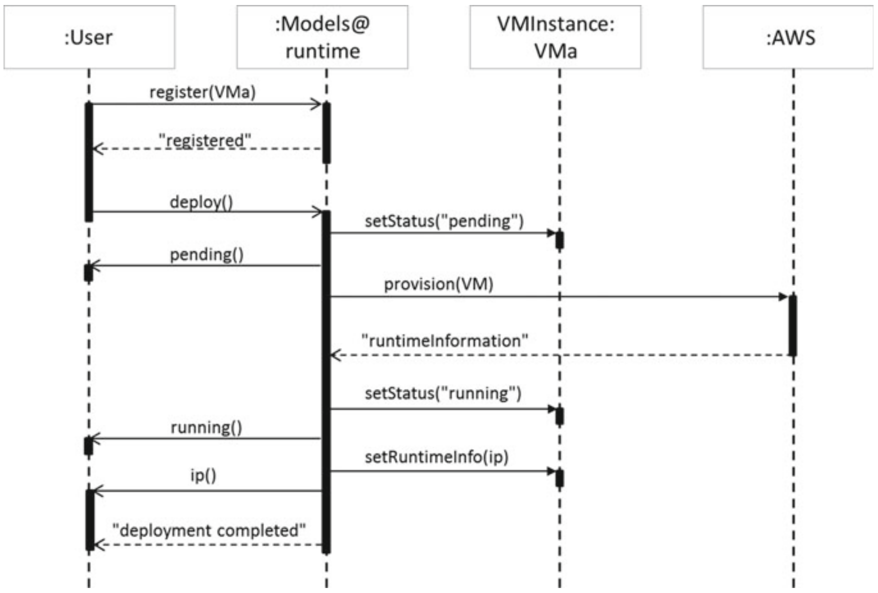

Fig. 9.6 Models@Runtime notification mechanism

Figure 9.6 presents a typical usage of the notification mechanism. First a client use an asynchronous command to register for being notified when a change occur on a specific VM. Then she exploits another asynchronous command to initiate a deployment. As a result, the Models@Runtime engine (i) changes the status of the object in the model that represents this VM to `pending` and sends a message that depicts this change to the client, and (ii) initiates the actual provisioning of the VM. Once terminated, the status of the VM is changed to `running` and the corresponding notification is sent. In addition, the Models@Runtime engine retrieves from the provider and populate the model with a set of runtime information such as the IP of the VM. For each of these changes in the model a notification is sent.

Currently, the communication with third-parties is achieved using the WebSocket protocol[1] in order to enable light-weight communications. Events are encoded as plain text and we provide a domain-specific language to define them, including the text format, the query and criteria to locate the relevant model element, the modification or change on the element, and the combination of other events. We defined the standard MOF (Meta-Object Facility) reflection modifications as the primitive events, and allow developers to further define higher level events as the composition of primitive ones. Using this language, one can also define the model changes on an abstract model as the composition of events on a concrete model, and

[1]http://www.websocket.org/.

in this way, it can be used as an event-based transformation. After each adaptation, the engine wraps the modification events into one message and send it to the WebSocket port.

In order to handle concurrency (i.e., adaptation actions coming from several third-parties), the Models@Runtime uses a simple transaction-based mechanism. The WebSocket component creates a single transaction which contains all the modifications from a third-party, and passes it to a concurrency handler. The handler queues the transactions, and executes them one after another without overlapping. Since all the modifications are simply assignments or object instantiation commands on the model in the form of Java objects, the time to finish a transaction of events is significantly shorter than the adaptation process.

9.4 Related Work

In the Cloud community, several solutions support the deployment, management and adaptation of Cloud-based application. However, to the best of our knowledge, none of them provides the same concepts and abstractions at runtime for the operators as applied by the developers.

Advanced frameworks such as Cloudify,[2] Puppet[3] or Chef[4] provide capabilities for the automatic provisioning, deployment, monitoring, and adaptation of Cloud systems without being language-dependent. Such solutions provide DSL to capture and enact Cloud-based system management. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [10] standard is a specification developed by the OASIS. TOSCA provides a language for specifying the components comprising the topology of Cloud applications along with the processes for their orchestration.

In addition, several approaches focus on the management of application based on PaaS solutions. Sellami et al. [11] propose an model-driven approach for PaaS-independent provisioning and management of Cloud applications. This approach includes a way to model the PaaS application to be deployed as well as a REST API to provision and manage the described application. The Cloud4SOA EU project [12] provides a framework for facilitating the matchmaking, management, monitoring and migration of application on PaaS platforms.

By constrast with the Models@Runtime engine, in all these approaches, the resulting models are not causally connected to the running system, and may become irrelevant as maintenance operations are carried out. The approaches proposed in the CloudScale [13] and Reservoir [14] projects suffer similar limitations.

[2]http://www.cloudifysource.org/.

[3]https://puppetlabs.com/.

[4]http://www.opscode.com/chef/.

On the other hand, the work of Shao et al. [15] was a first attempt to build a models@runtime platform for the cloud, but remains restricted to monitoring, without providing support for configuration enactment. To the best of our knowledge, the CloudMLModels@Runtime engine is thus the first attempt to reconcile cloud management solutions with modelling practices through the use of models@run-time.

9.5 Conclusion

In this chapter we presented how the MODAClouds Models@Runtime approach leverage upon MDE techniques and methods at runtime to support the continuous design and deployment of multi-Cloud applications. This includes support for their dynamic provisioning, deployment and adaptation by third party entities. Thanks to the proposed approach it is possible to exploit the same concepts and language for deployment and resource provisioning at both development and operation time. This facilitates interaction between developer and operation teams and helps reducing the gap between the two related activities as advocated by the DevOps movement.

References

1. Httermann M (2012) DevOps for developers. Apress
2. Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley Professional
3. Fitzgerald B, Stol KJ (2014) Continuous software engineering and beyond: trends and challenges. In: Proceedings of the 1st international workshop on rapid continuous software engineering. ACM, pp 1–9
4. Petcu D (2014) Consuming resources and services from multiple clouds. J Grid Comput 1–25
5. Ardagna D, Di Nitto E, Casale G, Petcu D, Mohagheghi P, Mosser S, Matthews P, Gericke A, Balligny C, D'Andria F, Nechifor CS, Sheridan C (2012) MODACLOUDS, a model-driven approach for the design and execution of applications on multiple clouds. In: ICSE MiSE: international workshop on modelling in software engineering. IEEE/ACM, pp 50–56
6. Blair G, Bencomo N, France R (2009) Models@run.time. IEEE Comput 42(10):22–27
7. Ruscio DD, Paige RF, Pierantonio A (eds) Special issue on success stories in model driven engineering 89(Part B) Elsevier (2014)
8. Morin B, Barais O, Jézéquel JM, Fleurey F, Solberg A (2009) Models@Run.time to support dynamic adaptation. IEEE Comput 42(10):44–51
9. Ferry N, Song H, Rossini A, Chauvel F, Solberg A (2014) CloudMF: applying MDE to tame the complexity of managing multi-cloud applications. In: Proceedings of UCC 2014: 7th IEEE/ACM international conference on utility and cloud computing
10. Palma D, Spatzier T (2013) Topology and orchestration specification for cloud applications (TOSCA). Technical report, Organization for the Advancement of Structured Information Standards (OASIS)
11. Sellami M, Yangui S, Mohamed M, Tata S (2013) PaaS-independent provisioning and management of applications in the cloud. In O'Conner L (ed) CLOUD 2013: 6th IEEE international conference on cloud computing. IEEE Computer Society, pp 693–700
12. Cloud4SOA EU project. http://www.cloud4soa.com

13. Brataas G, Stav E, Lehrig S, Becker S, Kopčak G, Huljenic D (2013) CloudScale: scalability management for cloud systems. In: ICPE 2013: 4th ACM/SPEC international conference on performance engineering. ACM, pp 335–338
14. Rochwerger B, Breitgand D, Levy E, Galis A, Nagin K, Llorente IM, Montero R, Wolfsthal Y, Elmroth E, Cáceres J, Ben-Yehuda M, Emmerich W, Galán F (2009) The reservoir model and architecture for open federated cloud computing. IBM J Res Dev 53(4):535–545
15. Shao J, Wei H, Wang Q, Mei H (2010) A runtime model based monitoring approach for cloud. In: CLOUD 2010: 3rd IEEE international conference on cloud computing. IEEE Computer Society, pp 313–320