Using Adaptation Plans to Control the Behavior of Models@runtime

Maksym Lushpenko, Nicolas Ferry, Hui Song, Franck Chauvel, Arnor Solberg

Department of Networked Systems and Services, SINTEF, Oslo, Norway
firstname.lastname@sintef.no

Abstract The models@runtime pattern proposes to leverage models as executable artefacts. A runtime model describing the state of the system is causally connected to the running system. Models@runtime engines typically play an active role in the definition of the adaptation plan that specifies the set of concrete tasks describing how the system should be adapted to reflect the change made on the runtime model. However, the generation of such plans generally relies on a set of predefined rules and the resulting plan is thus arbitrarily derived. In this paper we present an evolution of the models@runtime pattern with: *(i)* a DSL for the specification of adaptation plans and *(ii)* a runtime environment to enact such adaptation plans. The proposed approach has been applied to the Cloud Modelling Framework (CLOUDMF).

Keywords: models@run.time, dynamic adaptation, adaptation plan, cloud computing

1 Introduction

Models@runtime [1,2] is an architectural pattern for dynamically adaptive systems that leverages models as executable artefacts supporting the execution and adaptation of a system. This pattern proposes to decouple the internal state of the system from the management API used to modify it. A model describing the state of the system, hereafter called runtime model, is causally connected to the running system by exploiting its management API. Hence, any change in the running system is automatically reflected into the model, and, conversely, any change made to the model is enacted, on demand, onto the running system. This decoupling facilitates simulation, planning and automation of adaptation activities.

As part of the causal connection, the engine that synchronizes the runtime model with the running system automatically realizes the high-level adaptations made on the model into the running system. Yet, there are often alternative ways to enact these adaptations onto the system, which may significantly affect the effectiveness as well as the performance and the quality of service of the running system. Existing synchronization engines in projects such as DiVA [3], Kevoree [4], CLOUDMF [5], Genie [6] all implicitly define an adaptation plan that specifies the set of concrete actions describing how the system should be adapted to reflect the change made on the runtime model. This plan is arbitrarily derived from the difference between the desired and the current state of the system, and therefore overlooks more relevant options. In complex systems,

where guarantees in the quality of services are major concerns, the synchronization must allow for customization of the adaptation plan.

In this paper, we present our approach to support the dynamic modification of adaptation plans. We propose: *(i)* a domain specific modelling language for the specification of adaptation plans and, *(ii)* a runtime environment to support the enactment of such adaptation plans. The proposed approach has been implemented and applied to the Cloud Modelling Framework (CLOUDMF) [7,5].

The remainder of the paper is organized as follow. Section 2 presents a motivating example that will be used throughout the paper. Section 3 introduces the overall approach. Section 4 presents the modelling language before Section 5 describes the supporting runtime environment. Finally Section 7 compares the proposed approach with the state-of-the-art and Section 8 draws some conclusions.

2 Motivating Example

CLOUDMF supports developers and operators in managing multi-clouds systems that run accross several clouds and exploit both IaaS and PaaS solutions. CLOUDMF implements the models@runtime pattern and thus maintains a runtime model synchronized with the running system.

Such runtime model is described using a domain-specific modelling language (DSML) and includes the cloud environment (*e.g.*, virtual machines, applications servers or third-party services) as well as the software components of interest (*e.g.*, services, applications or libraries). This DSML, called the Cloud Modelling Language (CLOUDML), is inspired by component-based approaches [8]. In this respect, cloud deployment can be regarded as assemblies of components exposing ports, and bindings between these ports. Their life-cycle reflects typical operation activities such as uploads, installations, configurations, start or stop. Additional resources such as scripts, binaries or configuration files can complement the components and explicit the behaviour of each operation activities.

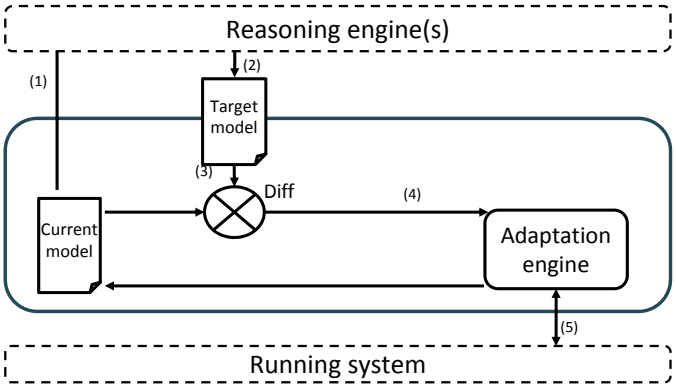

Figure 1. The CLOUDMF models@runtime architecture

Figure 1 depicts the architecture of the CLOUDMF models@runtime environment. The reasoning system reads the current runtime model (a CLOUDML model) (step 1), which describes the running system, and produces a target model (step 2). Then, the runtime environment computes the difference between the runtime and the target model (step 3) and generates an ordered list of adaptation actions. The synchronization engine traverses this list and triggers each action, thereby gradually adjusting the running system. The concrete actions may be either natively supported (*e.g.*, provisioning and upload) or delegated to the associated resources (*e.g.*, installation, configuration, start or stop).Details on how these steps are performed can be found in [5].

To illustrate the need for dynamic modification of adaptation plans, we discuss the deployment of a distributed real-time computation system called Apache Storm[1]. The deployment topology of Apache Storm is shown in Figure 2. It consists of a master node (Nimbus) which assigns tasks to slave nodes (Supervisors) whilst the coordination between the master and slave nodes is done by a Zookeeper[2] cluster which provides support for reliable distributed coordination.

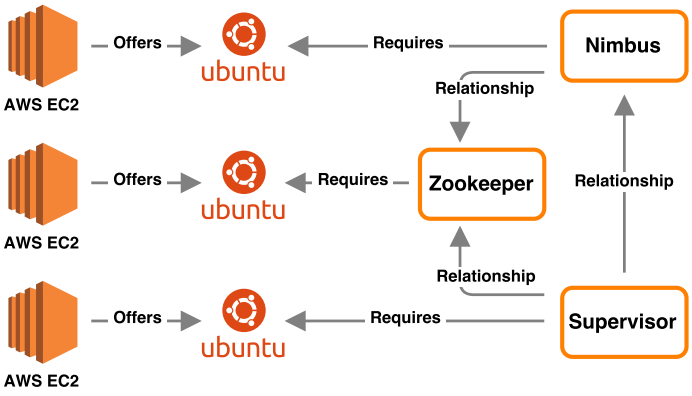

Figure 2. A Storm topology

In a manual deployment of Storm, an operator first configures the Nimbus and the Supervisors before she connects them. Indeed, their configuration yields files that are then filled in with specific data during their connection. By contrast, in a deployment automated with CLOUDMF, the default ordering of actions is the opposite: the connection preceeds the configuration. Even if workarounds are possible (*e.g.*, attach configuration script to the install command), they introduce two issues: *(i)* developers must know in advance the exact order in which all commands from all components are going to be executed by the CLOUDMF deployment engine, *(ii)* it can hinder the proper reuse of the component because of the misuse of install/configure commands.

[1] `https://storm.apache.org/`
[2] `https://zookeeper.apache.org/`

3 Overall Approach

In order to address these issues, we apply the approach depicted in Figure 3. First, the engine derives a tentative adaptation plan from a target model describing the desired system state. The user or a reasoning engine can then adjust it. Applied to our motivating example, once a deployment plan is generated from the defined Storm CLOUDML model, the user can then analyze and change it to trigger the configure commands before the connect commands. This consolidated adaptation plan is then fed in the adaptation engine, which is responsible for its execution.

When the runtime model is already synchronized with the running system, the engine first compares the actual and the desired system states and then derives a tentative adaptation plan, which can be modified before being enacted.

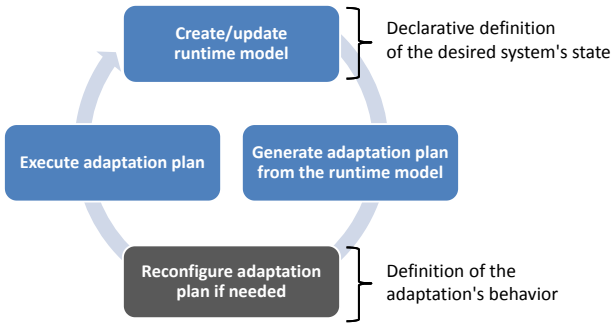

Figure 3. Overall approach

4 A DSL for adaptation plans

We model adaptation plans as workflows. Although existing "models@runtime" platforms usually derive adaptation plans as a sequence of adaptation actions, these actions may often be executed in parallel. For instance, in CLOUDMF, adaptation actions have to be parallelized as the associated running system is inherently distributed and deployment actions are often time consuming.

Our language to model adaptation plans is evolved from a subset of the concepts of UML activity diagrams. Figure 4 depicts the adaptation plan specifying the deployment of a simple Apache Storm cluster using our language. This workflow reflects the synchronization needed to properly provision, deploy and configure the whole Storm cluster. It starts by provisioning the cloud resources needed for the storm and the ZooKeeper clusters. Once the ZooKeeper cluster is installed and started, the installation of the Nimbus and its Supervisor is initiated and the workflow proceeds with connecting all these pieces.

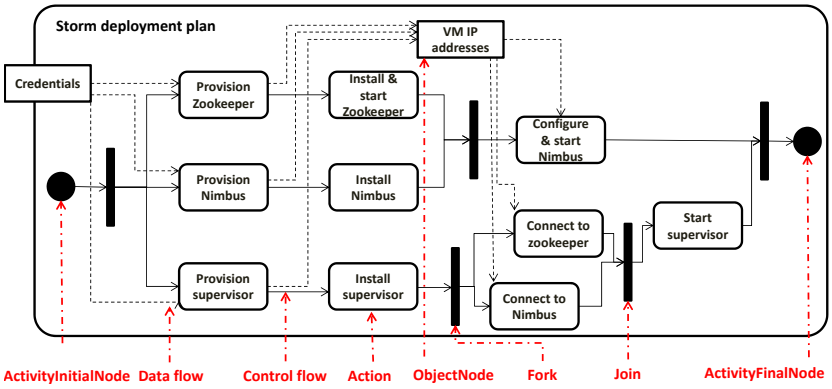

Figure 4. Simplified deployment plan of a storm cluster

Figure 5 shows the metamodel of our language. An adaptation plan consists is an `Activity` made of `ActivityNodes`, and `ActivityEdges`. An `ActivityNode` can be an `Action` to be performed on the running system. For instance, in the context of CLOUDML, it represents deployment actions such as the provisioning of a virtual machine (VM). An `ObjectNode` is another type of `ActivityNode` which can be used to store data that can later be exploited by `Actions` (*e.g.*, the IP address of the provisioned VM). Finally, an `ActivityNode` can be a `ControlNode`, which can be specialized into an `ActivityInitial` or `ActivityFinal` node describing the beginning and end of an adaptation plan or into a `join` or `fork` node to parallelize and synchronize the execution of independent `Actions`. An `ExpansionRegion` can be used to specify that a set of tasks has to be executed several times in parallel or sequentially. `ActivityNodes` are linked through `ActivityEdges`. The attribute `objectFlow` is used to specify that the edge represents either the control flow orchestrating the execution of `ActivityNodes` or the data flow to exchange objects between tasks.

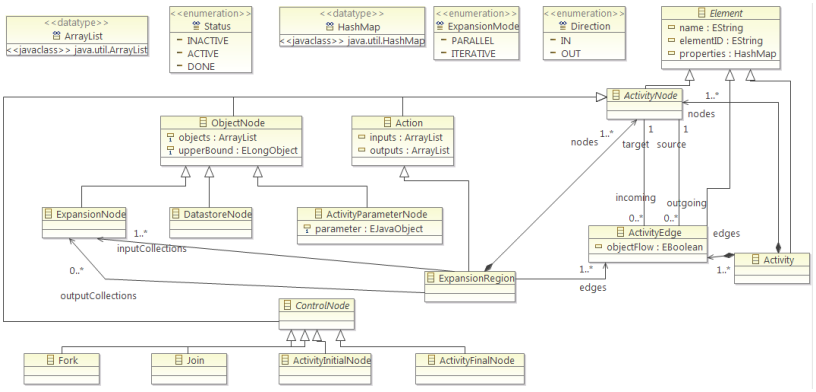

Figure 5. Adaptation plans metamodel

5 Runtime environment

This language is exploited by the runtime environment to specify and manipulate adaptation plans. In order to implement the approach presented in Section 3 we evolved the classical models@runtime architecture as depicted by the grey boxes in Figure 6. Once a target model has been provided as input (step 3) to the models@runtime environment, it can be compared to the runtime model if there is one. By contrast with the classical approach, the target model or the result of this comparison is then fed to an adaptation plan generator (step 4), which produces an initial adaptation plan (step 5). This plan is exposed by the models@runtime environment and can be modified by third parties. Once the appropriate plan identified, the latter can be validated (step 6) before being executed (step 7). The execution engine can then trigger a set of atomic actions (step 8) to be enacted onto the running system (step 9).

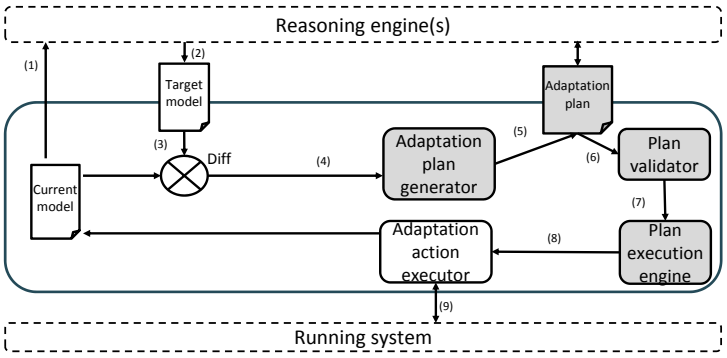

Figure 6. Evolution of the models@runtime architecture

In the following section we present our engines: *(i)* to generate an adaptation plan from a CLOUDML model and *(ii)* to execute the generated plan.

5.1 From CLOUDML models to Deployment plans

The engines to generate adaptation plans are typically specific to the domain of the models@runtime engine. In the context of our motivating example we created a transformation generating an adaptation plan from a CLOUDML model or from the comparison of two CLOUDML models. The plans are generated using the language presented in Section 4.

This transformation consists of the following four rules that are executed sequentially. During the execution of each rule, a set of `ActivityNodes` and `ActivityEdges` is created, together with the data required for the execution of `Actions`.

Provision cloud resources This rule creates in the adaptation plan the actions responsible for the parallel provisioning of virtual machines or cloud services on a given cloud provider. A Fork node is created to enable parallel provisioning.

Install components For every software component, this rule creates an action for each of the upload, download and install commands associated to the life-cycle management of the component. In addition, if a component depends on another, it ensures that the related actions are created in the proper order (*i.e.*, required components are installed first).

Connect components This rule creates an action for each commands specified in the resources attached to the relationships. Moreover, it creates a Join node to synchronize install actions of the components involved in the relationship.

Start components This rule creates actions in the adaptation plan for the configure and start commands of every component.

The way adaptation plans are generated is specific to the domain of the models@runtime engine. However, the language and its runtime environment are generic and can be applied to different domains. In the next section we present our execution environment.

5.2 Execution engine

Once an adaptation plan has been generated and validated, the adaptation engine visits the plan in the proper order (taking into account parallelization and synchronization points) and executes every `Action`.

This engine relies on two libraries: the Java Reflection API[3] and the Java Fork/Join framework[4]. The first one is used to ensure the independence of both the language and the execution engine from the domain on which the models@runtime engine is applied. Each action within an adaptation plan refers to a method that will enact the adaptation, the execution engine thus exploits the reflection mechanism to trigger the call to the specified method. The second library offers support for the parallelization and synchronization of the adaptation actions. In particular, it provides support for starting multiple threads asynchronously and for joining them afterwards.

Thanks to this mechanism, the engine executes each parallel branch of the adaptation plan concurrently meaning that there are no time dependencies between tasks in different parallel branches. The adaptation engine creates a number of threads whenever it enters a `Fork` node (explicit fork) or an `Action` node with multiple outgoing edges (implicit fork), and joins these multiple threads whenever it reaches a `Join` node (explicit join) or an `Action` node with multiple incoming edges (implicit join).

While the adaptation plan is being visited, the execution engine tracks the status of each adaptation action and reflects this status in the adaptation plan. Each node and edge in the adaptation plan may be in one of three states: `Inactive`, `Active` and `Done` (*i.e.*, before the execution of the node/edge, during, and after the execution respectively). As a result, the adaptation plan becomes a runtime model of the adaptation process that co-evolves with the runtime model of the running system.

---

[3] `https://docs.oracle.com/javase/tutorial/reflect/`

[4] `https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`

6 Implementation

Our language is implemented as plain Java objects, and also exists as an internal DSL within the CLOUDMF API. As for our Storm example, a whole deployment plan is represented as an `Activity` (see Listing 1.1).

Listing 1.1. Creating a deployment plan

```
1 Activity deploymentPlan = ActivityBuilder.getActivity();
```

The minimal set of nodes required for an executable plan consists of one `StartNode`, one or several `ActionNodes` and one `StopNode` (see Listing 1.2). The first task that has to be performed during a deployment is the provisioning of the virtual machines. For each VM to provision, an action is created in the plan. Two parameters have to be defined when creating an action: *(i)* the name of the method that will be used to enact the action (*e.g.*, reflection will be used to call the "provision" method that exploits the cloud provider API to actually provision the VM); and *(ii)* the necessary information to perform the call.

Listing 1.2. Creating start, stop and action nodes

```
1 ActivityInitialNode start = ActivityBuilder.controlStart();
  Action provision = ActivityBuilder.actionNode("Provision",VM);
3 ActivityFinalNode stop = ActivityBuilder.controlStop();
```

Virtual machines can typically be provisioned in parallel, thus a `ForkNode` has to be created and connected to the provisioning actions. Similar operation has to be performed to synchronize the tasks after provisioning (see Listing 1.3). The Boolean parameter used when creating these nodes indicates if the operators are applied to the control or data flows. In this case, they are applied to the control flow.

Listing 1.3. Creating synchronization actions

```
1 Fork fork = ActivityBuilder.forkNode(false);
  ActivityBuilder.connect(fork, provision, true);
3 Join join = ActivityBuilder.joinNode(false);
  ActivityBuilder.connect(provision, join, true);
```

Finally, *ObjectNode*s can be created to store some data, such as the IPs of the provisioned VMs (see Listing 1.4). An `ObjectNode` can be specialized into two types: `DatastoreNode`, which does not allow to store duplicates or `ParameterNode`, which can be used to supply some data to the deployment plan in the beginning of its execution.

Listing 1.4. Creating an object node

```
ObjectNode IP = ActivityBuilder.objectNode("IP", Type);
```

The results from this work are available as an open-source project[5] implemented in Java, using Maven as the build tool, and have been fully integrated with CLOUDMF[6]. The execution engine exploits the Java reflection and the join/fork frameworks. In addition, an engine has been created to automatically generate a DOT file from an adaptation plan together with a web page for their runtime graphical visualization.

[5] `https://github.com/SINTEF-9012/cloudml/tree/Maksym`

[6] `https://github.com/SINTEF-9012/cloudml/tree/master`

7 Related Work

As explained before, projects such as DiVA [3], Kevoree [4], and Genie [6] that perform architecture-based software adaptation relies on architectures similar to the one presented in Section 2. From the difference between the runtime model and a target model, a comparison engine identifies the list of adaptation actions to be performed in order to reach the desired configuration. Similar approaches also exist at a lower level of abstraction such as [9] which provides support for updating Java Software at runtime. By contrast with our approach, these works do not include mechanism enabling the runtime orchestration and adaption of the list of adaptation actions.

The Sm@rt (Supporting Models@Runtime) framework [10] supports developers in constructing a runtime component model on top of legacy systems. Developers define a meta-model, which specifies the types of elements that compose the running system as well as their relationships. Developers provide annotations on this meta-model to specify the relation between the model operations (*e.g.*, creating an element, reading an attribute, or reset a connection) and the system's management API. From the two inputs, the Sm@rt generation framework automatically generates the synchronization engine that maintains the causal connection between the model and the running system. This framework provides developers with the ability to tune the association between models modifications and system's adaptation actions. However, these actions cannot be orchestrated at runtime.

The EUREMA [11] framework supports the design and adaptation of self-adaptive systems that may involve multiple feedback loops. In particular, EURESMA allows developers to model explicitly feedback loops by capturing their runtime models, their usage, the flow of models operations as well as the relationships between these models. These models are kept alive at runtime and can be evolved. This approach does not offer explicit support for the definition of adaptation plans, however, the adaptation and monitoring transformations that causally connect the running system and the runtime model could be modeled as a feedback loops thus making our work complementary.

8 Conclusion & Future Work

In this paper we presented an evolution of the classical models@runtime approach to support the dynamic management of the adaptation behavior of a models@runtime environment. The proposed approach consists of: *(i)* a DSL for the specification of adaptation plans (*i.e.*, the plans describing the set of adaptation actions to be performed in order to reach the desired application state); and *(ii)* a runtime environment to enact such adaptation plans.

Within our current implementation the engines to generate and validate adaptation plans are specific to CLOUDMF. In future work we will consider generalizing and applying our approach to other domains as well as extending it with an exception handling mechanism possibly that could be combined with transaction and rollback support. In particular, we plan to apply it to the software product line domain as a runtime engine to build and adapt a product from a set of features. Longer term future work will consist in building a similar framework for the monitoring mechanism embedded into typical

models@runtime engines. This would enable external entities to fully control the way a runtime model is synchronized with the running system.

Acknowledgement. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number: 318484 (MODAClouds), 317715 (PaaSage).

References

1. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@Run.time to Support Dynamic Adaptation. IEEE Computer **42**(10) (2009) 44–51
2. Blair, G., Bencomo, N., France, R.: Models@run.time. IEEE Computer **42**(10) (2009) 22–27
3. Morin, B., Barais, O., Jezequel, J., Fleurey, F., Solberg, A.: Models@ run. time to support dynamic adaptation. Computer **42**(10) (2009) 44–51
4. Nain, G., Fouquet, F., Morin, B., Barais, O., Jézéquel, J.M., et al.: Integrating iot and ios with a component-based approach. In: EUROMICRO-SEAA. (2010) 191–198
5. Ferry, N., Song, H., Rossini, A., Chauvel, F., Solberg, A.: Cloudmf: Applying mde to tame the complexity of managing multi-cloud applications. In: Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on, IEEE (2014) 269–277
6. Bencomo, N., Grace, P., Flores, C., Hughes, D., Blair, G.: Genie: Supporting the model driven development of reflective, component-based adaptive systems. In: ICSE, ACM (2008) 811–814
7. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In O'Conner, L., ed.: Proceedings of CLOUD 2013: 6^{th} IEEE International Conference on Cloud Computing, IEEE Computer Society (2013) 887–894
8. Szyperski, C.: Component Software: Beyond Object-Oriented Programming (2nd edition). Addison-Wesley Professional (2011)
9. Cazzola, W., Rossini, N.A., Al-Refai, M., France, R.B.: Fine-grained software evolution using uml activity and class models. In: Model-Driven Engineering Languages and Systems. Springer (2013) 271–286
10. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating synchronization engines between running systems and their model-based views. In: Models in Software Engineering. Springer (2010) 140–154
11. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with eurema. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **8**(4) (2014) 18