Towards Meta-adaptation of Dynamic Adaptive Systems with Models@Runtime

Nicolas Ferry, Franck Chauvel, Hui Song and Arnor Solberg
SINTEF, Oslo, Norway

Keywords: Model-driven Engineering, Models@runtime, Dynamic Adaptive Systems.

Abstract: A models@runtime environment keeps a model in synchrony with a running system, this way a reasoning engine adapts the system by modifying this model. Existing models@runtime environments typically fail to let the user control what concepts form the model nor how the model is synchronised with the running system. This is yet mandatory in uncertain environments that are open, dynamic and heterogeneous. In this position paper we evolve the classical models@runtime architectural pattern to address this issue, together with some initial implementation results.

1 INTRODUCTION

Modern software combines many dedicated components (containers, databases, front-ends) and becomes large scale, distributed, and dynamic *systems*. These systems must inevitably evolve due to bugs, new features, evolution of dependencies or new service-level objectives, *etc.* These evolutions call for dynamically adaptive systems (DAS) that withstand modification while running. However, the complexity of managing these DAS rapidly overwhelms IT teams, which must therefore automate most maintenance operations (Mainsah, 2002). Self-adaptive systems (de Lemos et al., 2010) promise to mitigate this issue by automatically adjusting their behaviour to their environment, but their design remains an open challenge that results in *ad hoc* solutions (de Lemos et al., 2010).

Model-driven engineering (MDE) helps design these self-adaptive systems using domain-specific models that focus on domain concepts and better isolate separate concerns (France and Rumpe, 2007). The models@runtime pattern (Morin et al., 2009) applies these MDE ideas to self-adaptive systems: The system thus maintains a domain-specific model of its state (tailored for a given adaptation) and any change made to this model is automatically synchronised with the running system. However, existing models@runtime approaches predefine one such synchronisation policy between the model and the running system. This prohibits any dynamic modification of the adaptation mechanisms (so called "meta-adaptation") and reduces our ability to control (i) what concepts are reflected into the runtime model and (ii) how a change in this model is synchronised with the running system.

In this position paper, we enhance the models@runtime architectural pattern to support meta-adaptation and present initial implementation results.

The remainder is organised as follows. Sec. 2 first introduces our running example. Sec. 3 illustrates the current limitations of the models@runtime practice. Sec. 4 then describes how we evolved the classical models@runtime architecture. Finally, Sec. 5 discusses selected related work before Sec. 6 concludes with our future research directions.

2 MOTIVATING EXAMPLE

We consider as a running example a cloud service broker that calls for more flexible models@runtime architectures.

This broker registers cloud services, their specification as well as information about their status. It includes a reasoning engine that helps prevent and cope with services' failure. When a service is likely to fail—say because of high CPU load—the broker recommends alternative services. When a service fails, the broker automatically replaces it by a known substitute. The broker maintains, at runtime, a feature model that details all the services and their substitutes.

Our broker uses CLOUDMF (Ferry et al., 2014)

Ferry N., Chauvel F., Song H. and Solberg A.
Towards Meta-adaptation of Dynamic Adaptive Systems with Models@Runtime.
DOI: 10.5220/0006225905030508
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 503-508
ISBN: 978-989-758-210-3

to replace services. CLOUDMF helps developers and operators deploy and manage cloud systems that run across multiple clouds. The CLOUDMF runtime model relies on CLOUDML, a domain-specific modelling language that describes cloud environments (virtual machines, applications servers or third-party services) as well as the software components they host (services, applications or libraries). The components' life-cycle captures recurrent operations activities such as uploads, installations, configurations, starts and stops. Additional resources such as scripts, binaries or configuration files can complement the components, making these activities explicit.

For instance, the broker exposes a service that collects raw sensor data, together with an engine that detects specific patterns in incoming data streams. When this engine fails, the broker automatically replaces it by a distributed real-time computation system called Apache Storm[1]. Any deployment of Apache Storm includes a master node (called Nimbus) that assigns tasks to slave nodes (called Supervisors). Nimbus and Supervisors are storm-specific concepts that do not belong to the previous runtime model and therefore require specific configuration operations conflicting with the default CLOUDMF deployment policy. Thus, this requires adapting how the modifications in the CLOUDML model are synchronised with the running system. In addition, in order to improve failure predictions, we update the metrics that are monitored. Instead of CPU usage, we aggregate CPU usage, response time, and network traffic into a new metric called "workload". This evolution requires not only to change the modelling concepts (*i.e.*, new attributes in existing concepts), but also to change the synchronisation between the model and the system (*i.e.*, how to reflect the system state into the attribute values).

3 THE MODELS@RUNTIME PATTERN

Models@runtime (Morin et al., 2009; Blair et al., 2009) is an architectural pattern that embeds models during execution to ease adaptation and reconfiguration. This pattern decouples the internal state of the system from the API used to modify this very state. A *runtime model* describes this state in a semantic-rich way, and is continuously synchronised with the running system using that management API. Hence, any change in the running system appears in the model, while conversely, any change made to the model impacts, on demand, the running system. Models@runtime facilitates simulation, planning and automation of adaptation activities by hiding the specific API details and representing the data with semantics.

The models@runtime pattern includes a system, a model and several transformations. A *system* is a runnable artefact, whose state is partially observable and controllable from the outside during execution. A *model* is one representation of that system's state, from a certain perspective. A *transformation* is a process that consumes one artefact and outputs another. Depending on what artefacts transformations consume and outputs, we defined four types of transformation:

1. **Adaptation.** Both source and target are models. This is the traditional model transformation. A special case, which is common in models@runtime approaches, is an endogenous transformation where the source and the target are models that describe the same system.
2. **Monitoring.** The source is a system and the target is a model. With the monitoring transformation, the model represents certain state or behaviour of the running system.
3. **Enactment.** The source is a model and the target is a system. The model governs the state of the system.

Fig. 1 depicts a typical models@runtime environment. The reasoning engine reads the current runtime model (Step 1), which describes the running system, and then specifies how to reconfigure it in a target model (Step 2). The runtime environment next computes the difference between these runtime and target models (Step 3) and generates a sequence of reconfiguration actions. The adaptation engine then triggers each action, thereby gradually adjusting the running system (Steps 4 and 5).

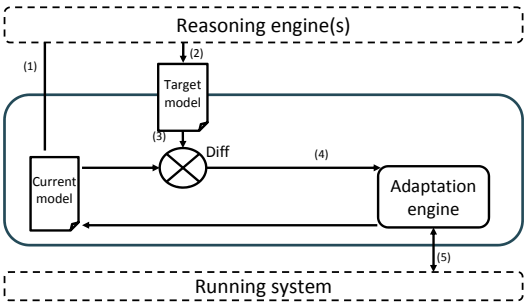

Figure 1: Typical models@runtime architecture.

However, the existing projects offering a models@runtime environment, such as DiVA (Morin

[1]See https://storm.apache.org/

et al., 2009), CLOUDMF (Ferry et al., 2014), Genie (Bencomo et al., 2008) and WComp (Ferry et al., 2009), do not permit correction of the behavior of the models@run-time environment: that is to influence what actions are triggered and in what order.

Monitoring Transformation Limitations. The monitoring transformation exploits the APIs of the system to build a model of its internal state, which may include its main components, their current status as well as the environment. While models@runtime help continuously adjust to a changing environment, not all changes can be foreseen at design-time and some will eventually exceed the adaptations abilities. In pervasive systems for instance, where mobile devices may join or leave at any time, we must dynamically extend the metamodel to capture new unforeseen types of devices. Besides, the API may offer different calls to retrieve data about these new devices. The models@runtime environment must therefore allow for both the customisation of its abstractions, and the customisation of the monitoring of these abstraction.

Enactment Transformation Limitations. The enactment transformation automatically propagates changes made to the target model onto the running system. Yet, these changes may generally be enacted in multiple ways, which may affect the performance and the quality of service (QoS) of the running system. Classical engines supporting one such enactment such as DiVA (Morin et al., 2009), CLOUDMF (Ferry et al., 2014), Genie (Bencomo et al., 2008), WComp (Ferry et al., 2009) all implicitly plan a sequence of concrete actions to adjust the system. This plan is arbitrarily derived from the difference between the desired and the current state, and may therefore overlook more relevant options. In complex systems, where QoS is a major concern, the enactment transformation must allow for customization of the adaptation plan.

In our example, an operator who manually deploys Storm would first configure the Nimbus and the Supervisors before to connect them. Indeed, their configuration yields files that she must then fill-in during their connection. By contrast, in a deployment automated with CLOUDMF, the default ordering of actions is the opposite: the connection precedes the configuration. However, CLOUDMF forbids modifying how such a change is enacted on the running system.

4 META-ADAPTATION

In order to overcome these limitations, we evolved the classical models@runtime architecture as depicted in Fig. 2. The green boxes relate to the enactment transformation, the orange boxes to the monitoring transformation, and the white boxes to the adaptation transformation. We detail below how we evolved the monitoring and enactment transformations.

4.1 Adaptable Monitoring

Observers gather specific information such as the status and properties of specific components in the system. When new entities join or leave, or when new properties have to be tracked, the set of observers is modified accordingly. Specific observers are responsible for observing the appearance and disappearance of new components in the running system and are called meta-monitoring components. Information from both the observers and the meta-monitoring components are sent to the Maintainer. The latter is responsible for combining the changes observed in the running system and updating the current model accordingly as well as for managing the set of observers.

At the current moment, we implemented the monitoring transformation in the context of the Broker@Cloud project by extending the DiVA framework (Morin et al., 2009). In particular, the framework offers the cloud service brokering platform presented in Sec. 2. Such brokering mechanism calls for adaptable monitoring transformation as the broker allows service providers to register new services, reset relationships between services, and introduce new types of property.

The proposed models@runtime environment encompasses two types of observers: complex-event processing observers and SPARQL observers. Complex-event processing observers exploit the broker@cloud publish-subscribe architecture. New events are published for any change in the broker (*i.e.*, new service for meta-monitoring) or in the registered service (*e.g.*, CPU usage). When a series of events conforms to a particular pattern, the observer creates a new event (*e.g.*, a workload event). The pattern and the new event are specified in the monitoring rules. SPARQL observers query the service specifications to deduce higher level information. These two types of observers generated more abstract knowledge from raw data. This is especially relevant when data available in the system APIs differs from what is needed in the runtime model.

We implemented the maintenance mechanism in

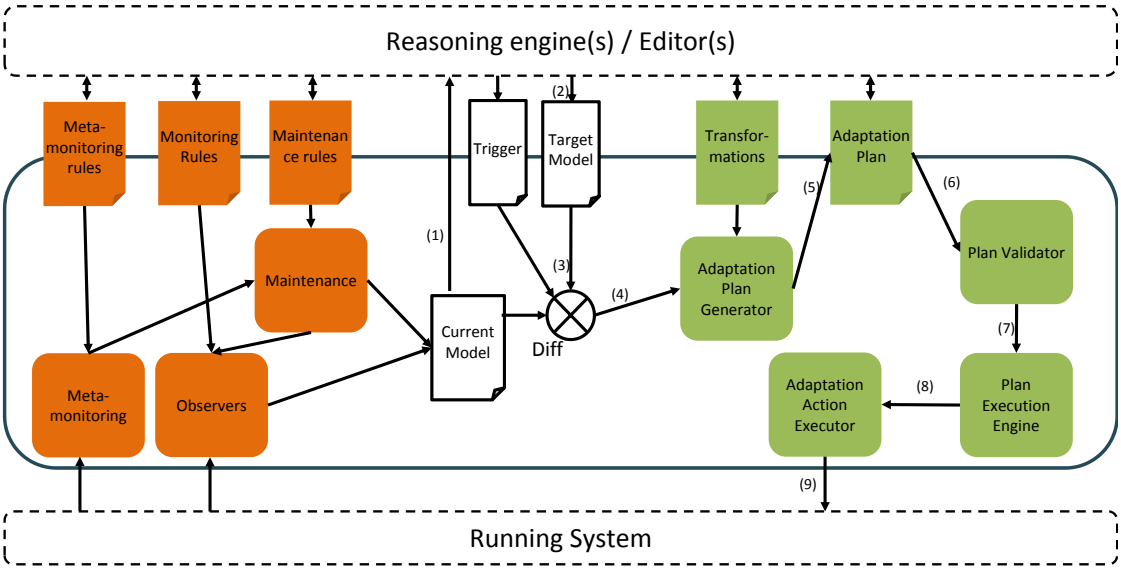

Figure 2: Evolved models@runtime architecture.

a metamodel-driven way. As Fig. 3 shows, we add annotations to the metamodel of the runtime model. These annotations specify how the changes of different kinds of elements, attributes, and references in the model relate to the system changes obtained via events or queries.

The maintenance mechanism coordinates all the annotations based on the following principles. 1) Once launched, the engine initiates an element in the root type. 2) When any event is captured, it looks for the classes with a `@create` annotation matching this event, and create an element for each of these classes. If the name of the new element already refer to an existing element, it simply merges them. 3) For any newly created element, it finds the `@query` annotation and trigger the query. 4) For any captured event, it looks through all the model elements whose type has an annotation matching the event. For each of these elements, it updates their corresponding attributes or references based on the event.

In future work, and in order to ease the control and adaptation of the maintenance mechanism, an aspect-oriented approach (Kiczales, 1996) could be exploited in order to dynamically weave the annotations into the metamodel of the runtime model.

```
1  class Variant{
2    @create on newsrv(?srvname)
3    @query qdim($srvname)
4    name: String {@id, @value=srvname}
5    for $p in owner.owner.properties:
6      @query qpval(name, $p.name)
7    @query qpval(name, $pname)
8      on newprop(?pname)
9    ref propvalue: PropValue* {
10     on qpvalo(!name, ?propname, ?v)
11     @value += PropValue{
12         name = $propname,
13         value = $v }
14     }
15     on impfailure(!name, ?value)
16     @value += PropValue{
17       prop="failurelikelihood",
18       value=?value}
19     }
20 }
21 class Dimension{
22   @create on qdim(?srvname, ?modname)
23   name: String{@id, @value=$modname}
24   ref variant:Variant{
25     @value += Variant{name=$srvname}}
26 }
```

Figure 3: Meta-model-driven specification of monitoring transformation.

4.2 Adaptable Enactment

The enactment transformation implements the following approach. First, the models@runtime environment derives a tentative adaptation plan from a target model describing the desired system state. The user or a reasoning engine can then adjust it. This consolidated adaptation plan is then fed in the adaptation engine, which is responsible for its execution. In case the runtime model is already synchronised 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.

We propose the process shown by the green boxes in Fig. 2. Once a target model is available (Step 3), it is compared to the current model (if there is one). By contrast with the classical approach, the result of this comparison, the diff (or the target, if there is no current model), is then fed to an adaptation plan generator (Step 4), which applies predefined rules to produce an initial adaptation plan (Step 5), open for modifications (manual or automated). Once this plan is confirmed, it can be validated (Step 6) before the execution engine then triggers its atomic actions (Steps 7 and 8) to adapt the running system (step 9). As a result, this approach enables controlling and tuning the adaptation plans as well as the engine that is responsible for generating adaptation plans.

Engines that generate adaptation plans typically depend on the domain of the models@runtime engine.

Currently, we have implemented the enactment transformation in the context of the CLOUDMF project (Ferry et al., 2014)[2]. In our motivating example, we created a transformation that generates an adaptation plan from a CLOUDML model or from the comparison of two CLOUDML models. Contrary to the adaptation plan, this transformation currently cannot be changed at runtime.

Our language to model and edit adaptation plans reuses a subset of the UML 2.0 activity diagrams. Fig. 4 depicts the adaptation plan that deploys our sensor data storage application. This language and its execution engine are generic and apply to different domains.

Once a valid adaptation plan is ready, the adaptation engine executes every `Action` following parallel branches and synchronisation. This engine relies on the Java Reflection mechanism to ensure independence of both the language and the execution engine from the domain on which the models@runtime environment is applied. Each action within an adaptation plan refers to a method that will enact the adaptation, the execution engine thus uses reflection to invoke the specified method.

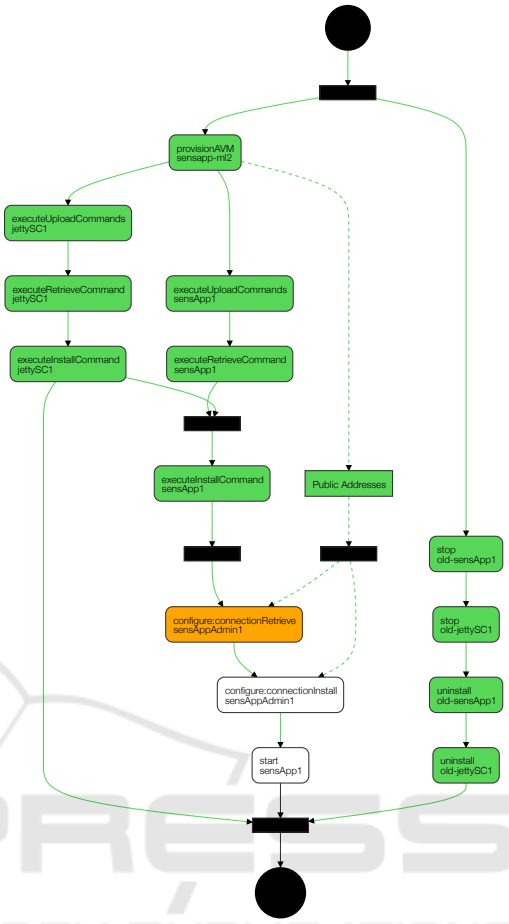

Figure 4: Adaptation plan.

5 RELATED WORK

Projects such as DiVA (Morin et al., 2009), and Genie (Bencomo et al., 2008), CLOUDMF (Ferry et al., 2014), which adapt software architectures, all rely on the classical models@runtime described in Sec. 3. From the difference between the runtime and target models, a comparison engine identifies what sequence of adaptation actions will reach the desired configuration. Similar approaches also exist at a lower level of abstraction such as (Cazzola et al., 2013), which updates Java Software at runtime. As opposed to our approach, this work does not enable the runtime orchestration and adaptation of the list of adaptation actions.

For the enactment transformation, Kevoree (Fouquet et al., 2012) exploits the concept of adaptation primitives that follows the type-instance pattern and offers a mechanism for dynamically adding/removing such primitives, e.g., defining how to deploy a module.

The Sm@rt (Supporting Models@Runtime) framework (Song et al., 2010) helps developers build a runtime component model on top of legacy systems. Developers first define the metamodel that specifies the types of elements that compose the running system and their relationships. Next, they annotate it with the relationships between the model operations (*e.g.*, create, get or set) and the system's management API. Sm@rt then automatically generates the engine

[2]Available at https://github.com/SINTEF-9012/cloudml/

that synchronises the model and the running system. Although developers can tune how models updates relate to the system's adaptation actions, these actions cannot be orchestrated at runtime.

The EUREMA (Vogel and Giese, 2014) framework supports the design and adaptation of self-adaptive systems that may involve multiple feedback loops. Developers explicitly model these feedback loops, 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 controlling the monitoring and enactment transformations. These transformations could however be modelled as a feedback loops thus making our work complementary.

6 CONCLUSION AND FUTURE WORK

We presented above an evolution of the classical models@run-time pattern to better manage the behaviour of a models@runtime environment and, in particular, how the runtime model is synchronised with the running system. This requires control over both the monitoring and enactment transformations. We also described an initial implementation of these mechanisms.

In future work we will focus on developing a single models@runtime environment that fully implements the architecture depicted in Fig. 2, while also handling errors and transactions. In addition, we plan to apply the mechanism to multi-layered architecture. Multi-layered architectures (Sykes et al., 2008) provide self-adaptive systems with the ability to adapt themselve to unforeseen circumstances. They advocates that the layer $n+1$ reconfigures the layer n underneath, recursively until the running system. However, this multi-layered architecture does not fit the classical models@runtime approach. The limited control over monitoring and enactment prevents the self-adaptive system to change its own adaptation behaviour. The extensions we propose allows to overcome this issue.

ACKNOWLEDGEMENTS

This research has received funding from the European Community's FP7 (2007-2013) and H2020 Programs under grant agreement numbers: FoF-2015.680478 (MC-Suite) and 645372 (ARCADIA).

REFERENCES

Bencomo, N., Grace, P., Flores, C., Hughes, D., and Blair, G. (2008). Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE*, pages 811–814. ACM.

Blair, G., Bencomo, N., and France, R. (2009). Models@run.time. *IEEE Computer*, 42(10):22–27.

Cazzola, W., Rossini, N. A., Al-Refai, M., and France, R. B. (2013). Fine-grained software evolution using uml activity and class models. In *Model-Driven Engineering Languages and Systems*, pages 271–286. Springer.

de Lemos, R., Giese, H., Müller, H. A., and et al, M. S. (2010). Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In de Lemos, R., Giese, H., Müller, H. A., and Shaw, M., editors, *Software Engineering for Self-Adaptive Systems II - International Seminar*, volume 7475 of *LNCS*. Springer.

Ferry, N., Hourdin, V., Lavirotte, S., Rey, G., Tigli, J.-Y., and Riveill, M. (2009). Models at runtime: service for device composition and adaptation. In *Proceedings of the 4th International Workshop on Models@ run. time*.

Ferry, N., Song, H., Rossini, A., Chauvel, F., and Solberg, A. (2014). CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In *Proceedings of IEEE/ACM UCC*.

Fouquet, F., Morin, B., Fleurey, F., Barais, O., Plouzeau, N., and Jezequel, J.-M. (2012). A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 135–144. ACM.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society.

Kiczales, G. (1996). Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es).

Mainsah, E. (2002). Autonomic computing: the next era of computing. *Electronics Communication Engineering Journal*, 14(1):2–3.

Morin, B., Barais, O., Jezequel, J., Fleurey, F., and Solberg, A. (2009). Models@ run. time to support dynamic adaptation. *Computer*, 42(10):44–51.

Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., and Mei, H. (2010). Generating synchronization engines between running systems and their model-based views. In *Models in Software Engineering*, pages 140–154. Springer.

Sykes, D., Heaven, W., Magee, J., and Kramer, J. (2008). From goals to components: a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8. ACM.

Vogel, T. and Giese, H. (2014). Model-driven engineering of self-adaptive software with eurema. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):18.