An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability

Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, Gordon Blair

An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability*

Brice Morin[1], Franck Fleurey[2], Nelly Bencomo[3],
Jean-Marc Jézéquel[1], Arnor Solberg[2], Vegard Dehlen[2], and Gordon Blair[3]

[1] IRISA/INRIA Rennes, Equipe Triskell, Campus de Beaulieu, 35042 Rennes Cedex, France
[2] SINTEF, Oslo, Norway
[3] Computing department, InfoLab21, Lancaster University, LA1 4WA, United Kingdom

Abstract. Constructing and executing distributed systems that can adapt to their operating context in order to sustain provided services and the service qualities are complex tasks. Managing adaptation of multiple, interacting services is particularly difficult since these services tend to be distributed across the system, interdependent and sometimes tangled with other services. Furthermore, the exponential growth of the number of potential system configurations derived from the variabilities of each service need to be handled. Current practices of writing low-level reconfiguration scripts as part of the system code to handle run time adaptation are both error prone and time consuming and make adaptive systems difficult to validate and evolve. In this paper, we propose to combine model driven and aspect oriented techniques to better cope with the complexities of adaptive systems construction and execution, and to handle the problem of exponential growth of the number of possible configurations. Combining these techniques allows us to use high level domain abstractions, simplify the representation of variants and limit the problem pertaining to the combinatorial explosion of possible configurations. In our approach we also use models at runtime to generate the adaptation logic by comparing the current configuration of the system to a composed model representing the configuration we want to reach.

1 Introduction

Context aware software systems that can automatically adapt to changes in their environments play increasingly vital roles in society's infrastructures. The demand for adaptive systems appears in many application domains, ranging from crisis management applications such as disaster and power management, to entertainment and business applications such as mobile interactive gaming, tourist guiding and business collaborations.

However, constructing and executing adaptive systems are highly complex tasks facing several challenges. Adaptive software systems are typically deployed on distributed platforms consisting of heterogeneous computing devices. The target platforms for a single system can range from computer networks of any size to small portable devices, such as phones or PDAs. Furthermore, a system is composed of components with variable configurations that might have dependency relationships that need to be

* This work was partially funded by the DiVA project (EU FP7 STREP)

resolved during adaptation; thus, compounding the complexity. Better techniques for taming the complexity of adaptive software during development are needed. Another challenge in adaptive system construction and execution is the issue of combinatorial explosion. Adaptive systems are often developed by defining several variation points, which represents points in the software where different variants of the implementation might be chosen to derive the final system configuration. Resolving these variation points leads to an exponential growth in the number of possible system configurations. This presents a major problem, since reasoning on a huge number of configurations to find the best possible configuration for the current context becomes too time consuming when considering the often strict requirements to response times these systems face.

Abstraction is the most fundamental principle applied in software engineering to encounter a continuously wider range of problems and increasing complexity [20]. In Model Driven Engineering (MDE), abstractions and transformations between levels are used to manage complexity. For example, the Model Driven Architecture (MDA) specifies three abstraction levels; a Computation Independent Model (CIM) describes the environment and specifies requirements; a Platform Independent Model (PIM) describes the parts that do not change from one platform to another; and a Platform Specific Model (PSM) includes descriptions of platform dependent parts. Another principle that are commonly applied in software engineering to handle complexity are separation of concern. Aspect Oriented Modeling (AOM) approaches provides advanced mechanisms for separation of concern such as mechanisms for encapsulating crosscutting features and for composing crosscutting features to form integrated models [9, 13–15].

In this paper we present a new approach where we address challenges in adaptive system construction and execution by combining certain aspect-oriented and model-driven techniques. In particular we use:

- Aspect-Oriented Modeling techniques in order to tackle the issue of the combinatorial explosion of variants. AOM allows us to encapsulate distinct variation points into aspects which are separated from the base model of the system's functionality. Then, distinct aspects might be composed into the base model in order to obtain different configurations. This approach allows us to reason on a limited and linearly increasing number of aspects, thus avoiding the problem of combinatorial explosion seen in other approaches.
- Model-Driven techniques to automate and improve the creation of the reconfiguration script needed to make the running system evolve from one configuration to another. Currently the adaptation logic of adaptive middleware relies on the execution of low-level and hand-written reconfiguration scripts, which specify all the possible transitions between the configurations. As these scripts decide how systems - possibly critical to safety - are manipulated at runtime, they require rigorous validation. Such a process is both time consuming and, even worse, prone to human errors. Instead of manually writing these scripts, we apply model driven techniques to generate them by analyzing the different variants of the system. In addition we apply model driven principles to provide models at run time for managing the execution of the adaptation at a more abstract level. This enable us to provide abstractions fine-tuned towards the adaptation task, reducing complexity. Furthermore, it makes our approach applicable on many execution platforms since our models at runtime is provided as platform independent models.

The remainder of this paper is organized as follows. Section 2 introduces a running example and presents some background. Section 3 presents our methodology for managing dynamic variability. Section 4 details our approach using the running example. Section 5 presents related works and Section 6 provides our conclusion.

2 Motivating Example

This section presents a brief background on management and support for variability in the context of dynamic adaptive systems, based on [1]. We next discuss the limitations of this approach and the solutions we propose. The discussion is in the context of mobile computing environments applications which need to dynamically discover services from a wide range of options that may be unknown during design. Such kind of applications propose a simple yet powerful motivating example of systems that need support for dynamic variability.

Traditionally, variability management has focused on variability that is solved at predelivery time, i.e. from requirements to deployment. However, adaptive systems exhibit degrees of variability that depend on runtime fluctuations in their contexts. This kind of variability is called dynamic variability or runtime variability [1]. Reflective and adaptive middleware platforms offer powerful mechanisms to achieve dynamic variability to enable adaptation at runtime. These mechanisms allow programmers to hard-code reconfiguration scripts to dynamically transform one component-based configuration into another.

2.1 Dynamic service discovery for mobile applications

Mobile applications need to dynamically adapt according to changes in their operating contexts. Mobile devices such as PDAs, mobiles, or laptops are capable of detecting and notify the user about new available services according to his/her preferences. The complexity arises from the fact that mobile adaptive applications are expected to support unanticipated variants associated with user preferences and properties of operating contexts that inevitably will arise during execution. Furthermore, different designs for service discovery protocols (SDPs) have been proposed. Hence, it may not be possible to completely specify at design time user preferences, properties associated with the contexts, or which protocols will be used to advertise services in a given context execution.

[8] presents a solution to overcome the challenges posed by heterogeneous service discovery protocols. The solution offers a common core architecture that individual discovery protocols follow. Using the final architecture, discovery protocols can be implemented and dynamically plugged into the middleware platform. Hence, different service discovery protocols can be used to discover services advertised by heterogeneous platforms. Furthermore, different policies can be uploaded during runtime to dynamically update the behavior of the application. Using this solution, the service discovery interaction platform from our example can take different roles that individual protocols could assume:

-**User Agent (*UA*)** to discover services on behalf of clients,

-**Service Agent (*SA*)** to advertise services, and,

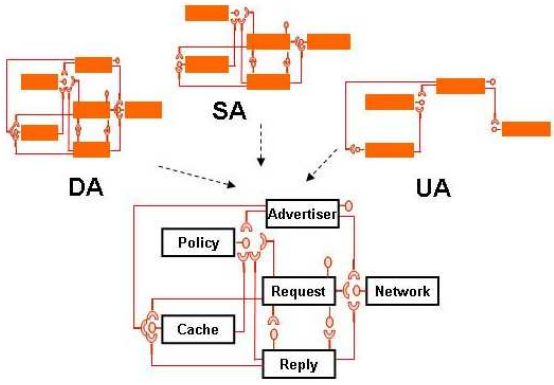

Fig. 1. Common Architecture and its Configurations

-**Directory Agent (*DA*)** to support a service directory where *SA*s register their services and *UA*s send their service requests. A DA also announces positive matches of requests against advertisements.

Depending on the required functionality, participating nodes might be required to support 1,2, or the 3 roles at any time. The common architecture, which is shown in Figure 1, has six components:

-Advertiser Component: used by *SA*s to advertise its services and by *DA*s to process incoming service advertisements storing them in cache. This component also deals with the maintenance of a directory overlay network.

-Request Component: used by *UA* and *DA*s to generate service requests.

-Reply Component: used by *UA*s and *DA*s to generate service replies.

-Cache Component: for common utility tasks such as management of temporary data, service advertisements, and location of neighboring directories.

-Policy Component: this component stores and deals with user preferences, application needs and/or inclusive context requirements.

-Network Component: for the transmission of messages.

Network, *Cache* and *Policies* components will always be present in any valid configuration. The other three components and their bindings will be part of the configuration or not, depending on the roles the protocol might perform (i.e. *SA*, *UA*, or *DA*). Hence, roles (agents) directly define the configuration (variants). Figure 1 shows three different configurations which are compliant with the architecture to support either a *UA* or *SA* role by restricting the number of components to only those required to provide the determined functionality. By using a complete framework configuration, a *DA* can also be supported.

In [1, 2], authors complement the solution shown above targeting the development of adaptive systems. This approach uses the Genie toolkit [1], to design stable runtime configurations, as well as the possible triggers that initiate the reconfigurations. In the case of the example above, the UA, SA, and DA are the possible configurations to be used by any specific SDP. At the end of the process, a state-machine model is produced where each state represents a configuration and each transition represents a conditional reconfiguration that transforms the source configuration into the target configuration.

Essentially, this state machine model drives the execution of the system. Using the Genie toolkit and the state machine model, designers can automatically generate XML files that represent the adaptation policies associated to transitions. Such policies can be dynamically introduce to change the behavior of the system during execution. Figure 2 illustrates the state machine model for the dynamic service discovery application and an example of a generated adaptation policy associated with the arc (pointed by the arrow from the top arc of the model to the generated policy). The approach has also been applied to consider configurations and reconfigurations associated with different domains such as routing protocols and networking technologies [2].

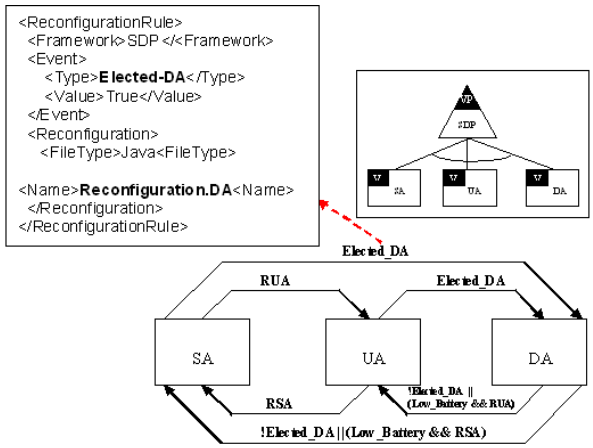

Fig. 2. Variants and transitions for the service discovery application

2.2 Limitations of existing approaches

The approach described in [1] presents several limitations. First, the possible system configurations need to be enumerated and fully specified. Secondly, the generated policies mainly specify the trigger events and which reconfiguration scripts have to be loaded to adapt the system from one state mode (i.e. agent role) to another. These scripts are currently hand-written using the API offered by the underlying execution platform. Finally, each state mode represents the whole system, in a given configuration. This is not enough in some cases. For example, in the case of the dynamic service discovery application described above, there is another variability dimension associated with the specific protocol to use such as ALLIA, GSD, SSD, SLP [8]. Each of these protocols has its own terms and rules. Therefore, in order to get a service agent and a user agent understanding each other, they need to use the same protocol. Taking into account different protocols increments the number of configurations and rules needed.

2.3 Contributions

The contribution of our proposed approach is twofold. Firstly, we argue that the reconfiguration scripts described above can also be inferred from the models, by comparing

the target configuration with the current configuration, for each transition that may be triggered during runtime. The results of this comparison will allow the dynamic generation of the corresponding reconfiguration, i.e. the identification of the components that should be added or deleted. This solution is possible as we proposed to keep a reference model that represents the current system and the possible modified model that is the result of the required adaptation. This will be detailed in the next sections. Secondly, using Genie, each state mode represents the whole system as one configuration per domain (in the example above the service discovery domain). As discussed above, this is not enough in some cases. Considering the complete enumeration of configurations for different variability dimensions, such as different protocol, may be an unmanageable task. In the following sections we will show how Aspect-Oriented Modeling techniques allow the separation and composition of different views of the system reducing complexity during the development.

3 Overview of the approach

The common practices in component-based dynamically adaptive systems (DAS) are to handle dynamic adaptation at the code level. The adaptation rules and the transformations that have to be performed on the running system are hard-coded and mixed with the code of the application [3, 5, 8]. This approach makes adaptive systems very difficult to understand, validate and evolve.

The idea of the approach we propose in this paper is to combine model driven and aspect-oriented techniques to handle the complexities of adaptive system construction and execution. Models cope with complexity through abstractions and are used both to specify the dynamic variability at design time and to manage run time adaptations. Aspect oriented techniques are utilized to model the adaptation concerns separately from the other aspects of the system. By utilizing model based abstractions and advanced separation of concerns in this way the adaptation becomes easier to design and understand, possible to validate and allows to easily evolve the adaptation policies even at runtime.

Figure 3 presents the conceptual model of the proposed approach. From a methodological perspective the approach is divided in two phases; design time and runtime.

At design-time, the application base and variant architecture models are designed and the adaptation model is built. At runtime, the adaptation model is processed to produce the system configuration that should be executed. The following paragraphs details the steps of Figure 3.

Because the potential number of configurations for an adaptive system grows exponentially with the number of variation points, a main objective of the approach is to model adaptive systems without having to enumerate all their possible configurations statically. To achieve this objective, an application is modeled using a base model which contains the common functionalities and a set of variant models which can be composed with this base model. The variant models capture the variability of the adaptive application. The actual configurations of the application are built at runtime by selecting and composing appropriate variants. An adaptation model specifies which variant have to be selected depending on the context of the running application.

The adaptation model is central to the approach as it captures all the information about the dynamic variability and adaptation of the adaptive system. It is built from

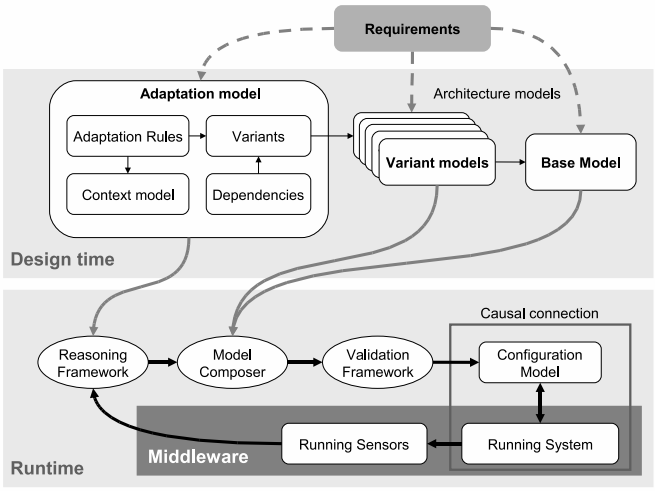

Fig. 3. Conceptual model of the approach

the requirements of the system, refined during design and used at runtime to manage adaptation. It is made of four main elements:

- **Variants**. This part of the model makes references to all the available variability for the application. Depending on the complexity of the system, it can be a simple list of variants, a data structure like a hierarchy or a complex feature model.
- **Dependencies**. The dependencies specify constraints on variants that can be used in a configuration. For example, the use of a particular functionality (variant model) might require or exclude others. These constraints reduce the total number of configurations by rejecting invalid configurations.
- **Context model**. The context model is a minimal representation of the environment of the adaptive application to support the definition of adaptation rules. We only consider elements of the environment relevant for expressing adaptation rules. These elements are updated by sensors deployed on the running system.
- **Adaptation rules**. These rules specify how the system should adapt to its environment. In practice these rules are relations between the values provided by the sensors and the variants that should be used.

During runtime appropriate configurations of the application have to be built from the base and variant models. To select the appropriate configuration, the reasoning framework processes the adaptation model and makes a decision based on the current context. The output of the reasoning framework is one or more options that match the adaptation rules and satisfies the dependency constraints. For each of these options the complete model of the corresponding configuration can be built at runtime using model composition.

Because the idea of the approach is to build configurations on demand rather than enumerating all configurations, each new configuration has to be validated at runtime.

The role of the validation framework is to process the configuration proposed by the reasoning framework in order to select the ones that are safe to deploy in the running system. The validation framework checks that the architecture model of the configuration is correct with respect to the constraints and protocols associated to the components it contains.

Once a configuration has been selected by the reasoning framework and checked by the validation framework, it can be deployed in the running system. To ease the adaptation of the running system, a model representing the system at a higher level of abstraction is causally connected to it. This model is transformed to match the configuration that has been selected for adaptation. The running system is adapted thanks to the causal connection. Because the connection goes in both directions, it also allows checking that the system is actually running the required configuration.

From a change to the environment quite a few steps are required to be able to safely adapt the system. These steps (especially model composition and validation) can require some time to execute and thus delay the actual adaptation. In practice, this issue is tackled by keeping track of the configurations that are validated and registering them for reuse. In extreme cases where predictable fast adaptation is required, a set of predefined configuration can be specified, built and validated in advance.

4 Managing the combinatorial explosion of configurations

The service discovery application described in section 2 has two different variability dimensions: the functionality and the network protocols. There are three variants for the functionality: UA, SA and DA (= UA and SA). Four variants of the protocols (ALLIA, GSD, SSD and SLP) can be supported separately or not. This leads to 45 configurations (2^4-1 protocols and 3 functionalities) and potentially 1980 (45x44) different reconfigurations. However, in the example of the service discovery application, the two variability dimensions and their aspects are independent and triggered by distinct events. Thus, it is possible to manage all the reconfigurations with 12 scripts, for respectively adding and removing each aspects.

4.1 Using AOM to represent the variability

To avoid the combinatorial explosion, we propose to model the variants instead of the configurations. This way, the number of models to be defined grows linearly with the variability. The configurations can then be built by automatically combining the variants. In practice this is achieved using Aspect-Oriented Modeling techniques for architecture models. The application commonalities, i.e. the architecture elements which are part of all configurations, are captured in a single base model. All the variants are then defined as aspect models to be woven in the base model. From a particular selection of variants, the corresponding configuration can be built automatically by weaving the corresponding aspect models into the base model.

The specific AOM technique we use is the SMARTADAPTERS approach [14, 15]. SMARTADAPTERS has formerly been applied to Java programs and UML class diagrams [14]. More recently, we have generalized this approach to any domain metamodel [15]. SMARTADAPTERS automatically generates domain-specific AOM frameworks using an input domain-metamodel. In this paper, the domain metamodel we use

is a generic component model representing the main concepts needed to describe the topology of running systems: components, binding, ports, *etc*.

In SMARTADAPTERS, an aspect is composed of three parts:*i)* a graft model, representing **what** we want to weave, *ii)* an interface model, representing **where** we want to weave the aspect and *iii)* a composition protocol specifying **how** to weave the graft model into the interface model. The graft model is a model fragment representing a given concern. The interface model is a model fragment parameterized by roles allowing the interface model to be matched in different base models. Finally, the composition protocol is described by model transformation primitives that manipulate elements from the graft and the interface models.

4.2 Application to the Service Discovery Example

For handling the functionalities of the service discovery, the application is separated into a base model and two aspects. The base model contains the common components: *Policy*, *Cache* and *Network*.

The first aspect corresponds to the user agent (UA) role and is illustrated in the left part of Figure 4. The graft model contains all the components and bindings needed to realize the functionality of the UA role. The interface model contains all the base components needed to integrate the graft model: *Policy*, *Cache* and *Network*. The composition protocol, represented by the interconnecting lines, specifies how to weave the graft model into the interface model. It consists in binding components of the graft model to components of the interface model, and vice-versa. Similarly, the second aspect corresponds to the service agent (SA) role and is illustrated in the right part of Figure 4.

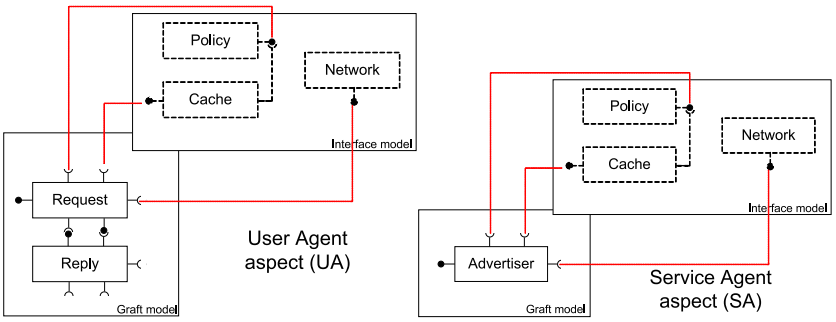

Fig. 4. User Agent and Service Agent aspect

These two aspects allow building the three functional configurations of the service discovery application. Weaving only the User Agent aspect leads to the User Agent configuration, weaving only the Service Agent aspect leads to the Service Agent configuration and weaving both aspects leads to the Discovery Agent configuration.

We have illustrated the approach using the variability on the functionalities of the application but the variability on the network protocols is handled similarly. Four aspects have to be defined for each of the four protocols and these aspects have to be

woven alternatively to build the corresponding configurations. As a result, the complete service discovery application is modeled using a base model and 6 aspects instead of the 45 models and/or the 12 scripts needed to specify all the configurations.

4.3 Discussion

Even with the simple example we described, our approach allows reducing by 50% the number of artifacts (aspects or scripts) required to describe the variability of the application. If the whole configuration models are required, (*e.g.,* for validation purpose) we reduce by 86% the number of models (6 aspects and 45 configurations) needed to describe the whole space of configurations. The important property is that the number of models grows linearly with the variability instead of exponentially when all configurations have to be described. It is interesting to also notice that the aspect models are smaller and simpler models than complete configuration models, as they focus on a single concern.

A consequence of the approach is that the adaptation rules do not have to select one specific configuration but sets of variants to include or exclude. In the service discovery example, if the adaptation is specified by a state machine the adaptation rules for the functionalities have to be duplicated for each network protocol. Using the proposed approach the adaptation rules for functionalities and network protocols can be defined separately since these two variability dimensions are independent.

In practice, engineers are used to state machines to represent and check adaptation policies. The proposed technique uses aspects and more generic rules which might be more difficult to understand and verify. This is especially true if there are complex interactions between the variability dimensions and variants. To overcome this limitation, the adaptation state matching of the application can be build automatically (completely or partially) from the specification of the variants and the adaptation rules. This gives an opportunity to the designer to check on his usual representation that the aspects and rules yield the expected adaptation.

5 Generating the adaptation

In this section, we present our model-driven causal connection responsible for reflecting changes from the model to the platform, and vice-versa. We illustrate our proposition on the service discovery application and discuss the advantages and limitations of our approach.

5.1 Using MDE to generate the adaptation logic

A key characteristic of adaptive systems is their ability to automatically adapt a running application. Our solution adopts a model-driven approach to runtime adaptation by using models at runtime, where the runtime model is connected to the executing system by a causal connection. This causal connection allows us to manage the executing system by manipulating its model. Our approach is similar to how reflection/reflective platforms work, however, using higher level models adds the benefit of remaining platform independent.

Figure 5 presents our approach to runtime adaptation and details the causal connection depicted in Figure 3. A reference model is generated using reflection over the

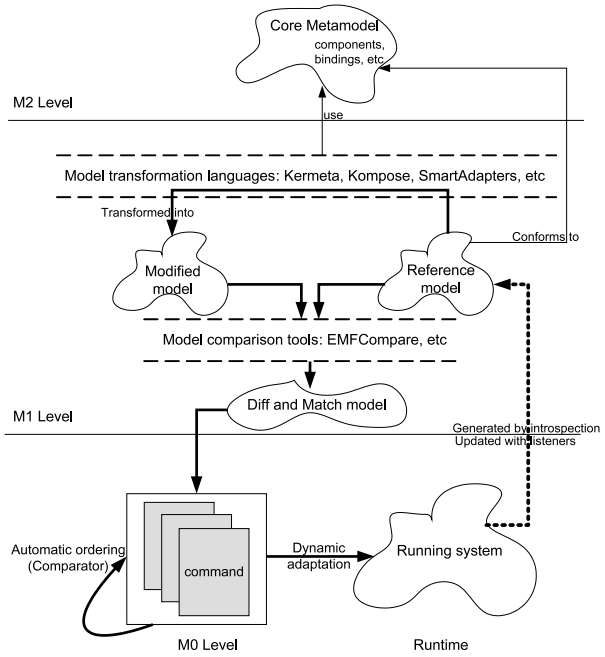

Fig. 5. Model-Driven adaptation at runtime

running system, representing the current running configuration. This reference model conforms to our core metamodel for representing component-based running systems. This metamodel contains the core concepts needed to represent a component-based configuration at runtime and is independent of any specific execution platform. The reference model is updated using listeners that observe the architectural reconfigurations of the running system, which allows us to update the model without instantiating it from scratch.

Adaptations are triggered by an adaptation need, usually caused by a context change. In our approach, a reasoning framework decides on a set of aspects according to the new context, and a new configuration the application should adapt to is created by weaving these aspects. It is also possible to create a new configuration by a model transformation or manually by modeling the modified model in a graphical editor.

Once the new configuration is created, it is compared with the reference model (representing the existing configuration). It produces a diff and a match model that specifies the differences and the similarities between the models. Note that this model comparison makes our approach independent from any particular transformation language. We browse both diff and match models to analyze the relevant changes between the reference model and the new configuration. During this analysis, we instantiate some reconfiguration commands, responsible for adding and/or removing bindings and/or components, etc. These commands are ordered according to their priority. Indeed, the way the model is transformed does not really matter, provided that the modified model still conforms to the metamodel after transformation whereas the running application should be

adapted rigorously. In the final step, the ordered sequence of commands is executed by the platform in order to actually adapt the running system.

As a verification step we check that the new reference model (which is automatically derived through reflection) is identical to the configuration model we wanted to reach. This is an important step as it allows us to verify that all the adaptation commands have been executed successfully.

5.2 Application to the Service Discovery Example

Let us assume that due to a context change, the application is to be adapted from the currently running User Agent configuration to the Service Agent configuration (Figures 5, 2 and 4). Looking at Figure 5, the former corresponds to the reference model while the latter corresponds to the new configuration. The model comparison detects several changes between these two configurations: *i)* the *Reply* and *Request*, as well as their bindings, have been deleted; *ii)* the *Advertiser* component is introduced into the runtime architecture and connected to the *Cache*, *Policy* and *Network* components. Based on this information, we automatically instantiate a set of reconfiguration commands in order to adapt the system at runtime. As an example, Listing 1.1 presents a reconfiguration script that was generated for the Fractal platform. This script is written in FScript [6], a language that abstracts the Fractal API.

Listing 1.1. UA to SA reconfiguration script expressed in FScript [6]

```
action reconfigureUAtoSA(root, Request, Reply, Advertiser){
  stop($Request);
  stop($Reply);
  // Remove the Request component
  unbind($Request/interface::network);
  unbind($Request/interface::cache);
  unbind($Request/interface::policy);
  unbind($Request/interface::reply);
  remove($root, $Request);
  // Remove the Reply component
  unbind($Reply/interface::network);
  unbind($Reply/interface::cache);
  unbind($Reply/interface::policy);
  unbind($Reply/interface::request);
  remove($root, $Reply);
  // Add the Advertiser component
  add($root, $Advertiser)
  bind($Advertiser/interface::network);
  bind($Advertiser/interface::cache);
  bind($Advertiser/interface::policy);
  start($Advertiser);
}
```

5.3 Discussion

Our approach of using causally connected models during runtime for dynamic reconfiguration has several advantages. First and foremost, we allow automatic generation

of reconfiguration scripts instead of having to write them by hand. In our example reconfiguration example from the previous sub-section, 17 operations are needed to reconfigure the application, as shown in Listing 1.1. These operations have to be ordered correctly to produce a consistent script. Our model-driven approach, based on model comparison, allows us to automatically compute and order the reconfiguration operations. Given that the transformations generating the reconfiguration scripts have undergone rigorous testing, these scripts will produce less errors than humans - thus increasing safety. In the service discovery application, we generate 6 scripts to handle the variability on the functionalities (one for each transition in Figure 2).

Second, we remain independent from any model transformation language. The model comparison allows us to be totally disconnected from any tools for manipulating models. Third, we validate target configurations before actually adapting the running system, by checking static constraints and simulating the models [4] for example using Kermeta [16]. It improves the confidence of runtime adaptation, especially when the underlying execution platform is not transactional and does not offer support for rolling back to the previous consistent configuration. In fact, our causal link is strongly synchronized from the running system to the model and delayed from the model to the running system. Finally, our approach can be mapped to different reflective execution platforms. For example, we can monitor and adapt Fractal [3] and OpenCOM [5] systems using the same kind of models.

While using a causally connected model to manipulate the running system has many benefits, it also has a cost. Response times are often important for adaptive systems, and calculating the diff model and automatically generating the reconfiguration scripts from it takes more time than simply executing predefined scripts. A possible solution to that problem is to pre-generate the critical scripts before the system execution. This way, the adaptation can be performed very quickly and the causaly conected model can be updated afterwards. Another solution to avoid model comparison would be to directly connect a particular model transformation language (*e.g.* SmartAdapters) to our causal link. This language would instantiate reconfiguration commands during model transformation and execute these commands after transformation. However, this would make our approach specific to a given model transformation language, while remaining independent from the execution platform.

6 Related Works

Recent middleware platforms like Fractal [3] or OpenCOM [5] propose ways to adapt a system at runtime, inspired by the work by Oreizy *et al.* [17] ten years ago. These approaches do not really propose to manage variability at runtime but propose mechanisms to reconfigure a system at runtime. We propose to map our causal link to any of these platforms. Our metamodel can be seen as a dynamic ADL to describe the running system. We can use any (aspect-oriented) model transformation languages to manipulate the reference model. We are not limited to an ad-hoc imperative reconfiguration language. Our causal link automatically computes the adaptation logic by comparing the reference model to a modified model.

Many mechanisms for runtime variability management have been proposed. They are mainly focused on exchange of runtime entities using parametrization, inheritance, and preprocessor directives [10, 18, 19]. Our approach is more coarse-grained and uses

architecture based models for the management of whole sets of components, their connections and semantics [1]. We can adapt running system via high level transformation languages and graphical editors.

Of particular relevance to our work is MADAM/MUSIC [7, 12] which uses the adaptation capabilities offered by middleware platforms, and treats dynamically adaptive systems as dynamic software product lines [11] with the corresponding support for variability management. The main variability mechanism consists in loading different implementations for each component type (primitive or composite) of the architecture. By decomposing the system into a base model and several aspects, we reduce the complexity related to the representation of variation points and their selection. We automatically compute safe reconfiguration scripts to adapt the system from the reference model to a target configuration. Finally our approach is generic: it can be mapped to different execution platforms whereas MADAM is limited to mobile computing applications, and we can use different model transformation languages to modify the reference model *e.g.* AOM languages like SmartAdapters [14, 15], MATA [13] or Kompose [9].

Wolfinger et al. [21] demonstrates the benefits of the integration of an existing product line engineering tool suite with a plug-in platform for enterprise software. As in our case, automatic runtime adaptation and reconfiguration are achieved by using the knowledge documented in variability models. Our differences exist mainly because of the different aims of each approach. Their work focuses on enterprise software while our work covers the domains grid, mobile computing, and embedded systems. While variability decisions in [21] are user-centered our variability decisions, based on the Genie approach [1] are environment-centered.

7 Conclusion and Future Works

In this paper we have presented a novel combination of Model-Driven Engineering (MDE) and Aspect-Oriented Modeling (AOM) to support dynamic variability. AOM allows us to focus on variability dimensions with no need to consider the whole configuration. By composing aspects, it is possible to produce a wide range of configuration models, while managing the combinatorial explosion of variants. Using a MDE approach, we use these configuration models to generate the adaptation logic needed to adapt the running system from one runtime configuration to another, instead of writing it by hand.

In future works, we plan to extend our core metamodel and monitor interesting properties of the running system, such as QoS-related properties, resource consumption, etc. This will allow us to develop a reasoning framework that will automatically select and weave the most adapted aspects. Another perspective is to pre-compile the most useful reconfiguration scripts. For the moment, our causal link computes these scripts at runtime. However, compiling the scripts implies that the model is not yet synchronized with the running system and consequently we do not know which components and bindings to adapt. Languages like FPath [6] may help us in retrieving the component we want to adapt at runtime.

References

1. N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *VaMoS'08: 2nd Int. Workshop on Variability Modeling*

of Software-intensive Systems, Essen, Germany, January 2008.
2. Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE 2008 - Formal Research Demonstrations Track*, 2008.
3. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.
4. F. Chauvel, O. Barais, J.M. Jézéquel, and I. Borne. A Model-Driven Process for Self-Adaptive Software. In *ERTS'08: 4th European Congress on Embedded Real Time Software*, Toulouse, France, 2008.
5. G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A Component Model for Building Systems Software. In *SEA'04: IASTED Software Engineering and Applications*, Cambridge MA, USA, November 2004.
6. P.C. David and T. Ledoux. Safe Dynamic Reconfigurations of Fractal Architectures with FScript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.
7. J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.
8. C.A. Flores-Cortés, G. Blair, and P. Grace. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad-hoc Environments. *IEEE Dist. Systems Online*, 2007.
9. R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing Support for Model Composition in Metamodels. In *EDOC'07: 11th Int. Enterprise Computing Conf.*, 2007.
10. M. Goedicke, C. Köllmann, and U. Zdun. Designing runtime variation points in product line architectures: three cases. *Science of Computer Programming Special Issue: Software variability management*, 53(3):353 – 380, 2004.
11. S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4), April 2008.
12. S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC'06: 10th Int. Software Product Line Conf.*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.
13. P.K. Jayaraman, J. Whittle, A.M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.
14. Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, October 2007.
15. B. Morin, O. Barais, and J. M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *VaMoS'08: 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, January 2008.
16. P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.
17. P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. In *ICSE'98: 20th Int. Conf. on Software Engineering*, Washington, DC, USA, 1998.
18. E. Posnak and G. Lavender. An adaptive framework for developing multimedia. *Communications ACM*, 40(10):43–47, 1997.
19. M. Svahnbergel, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705 – 754, 2005.
20. Tetsuo Tamai. Abstraction orientated property of software and its relation to patentability. *Information & Software Technology*, 40(5-6):253–257, 1998.
21. R. Wolfinger, S. Reiter, D. Dhungana, P.Grunbacher, and H. Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *ICCBSS'08: 7th Int. Conf. on Composition-Based Software Systems*, pages 21 – 30, 2008.