

Modeling and Validating Dynamic Adaptation¹

Franck Fleurey¹, Vegard Dehlen¹, Nelly Bencomo², Brice Morin³, Jean-Marc Jézéquel³

¹ SINTEF, Oslo, Norway

² Computing Department, Lancaster University, Lancaster, UK

³ IRISA/INRIA Rennes, Equipe Triskell, Rennes, France

Abstract. This paper discusses preliminary work on modeling and validation dynamic adaptation. The proposed approach is on the use of aspect-oriented modeling (AOM) and models at runtime. Our approach covers design and runtime phases. At design-time, a base model and different variant architecture models are designed and the adaptation model is built. Crucially, the adaptation model includes invariant properties and constraints that allow the validation of the adaptation rules before execution. During runtime, the adaptation model is processed to produce a correct system configuration that should be executed.

1 Introduction

In [7] we presented our work on how we combine model-driven and aspect-oriented techniques to better cope with the complexities during the construction and execution of adaptive systems, and in particular on how we handle the problem of exponential growth of the number of possible configurations of the system. The use of these techniques allows us to use high level domain abstractions and simplify the representation of variants. The fundamental aim is to tame the combinatorial explosion of the number of possible configurations of the system and the artifacts needed to handle these configurations. We use models at runtime [3] to generate the adaptation logic by comparing the current configuration of the system and a newly composed model that represent the configuration we want to reach. One of the main advantages is that the adaptation logic do not have to be manually written.

The adaptation model covers the adaptation rules that drive the execution of the system. These rules can be dynamically introduced to change the behavior of the system during execution. We also discussed in [7] the need of techniques to validate the adaptation rules at design time. In this paper we discuss our preliminary work on how to perform simulation and allow for model-checking in order to validate adaptation rules at design time. The model validated at design time is used at runtime.

The remainder of this paper is organized as follows. Section 2 presents an overview of our methodology for managing dynamic adaptation. Section 3 gives details on our meta-model for adaptive systems, and shows through a service discovery example how it can be used to model variability, context, adaptation rules and constraints. Section 4 shows how we simulate the adaptation model to validate the adaptation rules. Section 5 explains our solution for runtime model-based adaptation. Finally, Section 6 discusses the main challenges our work is facing and concludes.

¹ This work is done in the context of the European collaborative project DiVA (Dynamic Variability in complex, Adaptive systems).

2 Overview of the approach

Figure 1 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 to be used during execution. The following paragraphs details the steps of Figure 1.

Since 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. In order 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 the 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 the appropriate variants. The adaptation model does not deal with the basic functionality which is represented by the base model. Instead, the adaptation model just deals with the adaptive parts of the system represented by the variant models. The adaptation model specifies which variants should be selected according to the adaptation rules and the current context of the executing system.

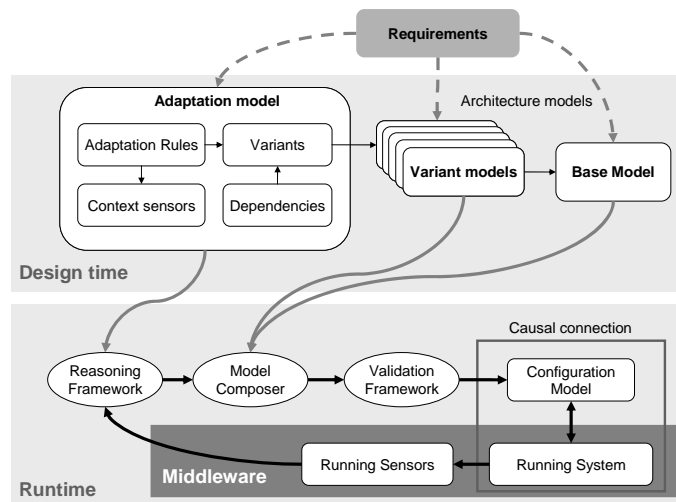


Fig. 1. Overview of the proposed approach

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 the requirements of the system, refined during design and used at runtime to manage adaptation. The adaptation model has four main elements:

- *Variants*: They make references to 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.
- *Constraints*: They specify constraints on variants to be used over 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*: 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.
- *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 are composed from the base and variant models. In order 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 a configuration that matches the adaptation rules and satisfies the dependency constraints. The model of this configuration can be built at runtime using model composition.

3 Adaptation Model

This section presents the adaptation meta-model and how it is applied to a Service Discovery Application (SDA). The SDA is a solution to tackle heterogeneity of service discovery protocols is presented in [5]. The solution allows an application to adapt to different service discovery protocols and needs during execution. The service discovery platform 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,
- Directory Agent (*DA*) to support a service directory.

Depending on the required functionality, participating nodes might be required to support 1, 2, or the 3 roles at any time. A second variability dimension is the specific service discovery protocols to use, such as ALLIA, GSD, SSD, SLP [5]. Each service discovery protocol follows its own rules. As a result, in order to get two different agents understanding each other, they need to use the same protocol [7]. These decisions have to be performed during execution.

The next sub-section presents an overview of the meta-model and the following sub-sections detail how it is instantiated for the SDA example.

3.1 Meta-model for variability and adaptation

As detailed in the previous section the adaptation model include four different aspects: *variants*, *adaptation rules*, *dependencies* and *context*. Additionally, links to the architecture models and concepts for rules and expressions are supplied. The meta-model is shown in Figure 2. As can be seen from the figure, colors are used to differentiate between the categories.

The colors indicate the following:

- Grey – base and aspect architecture models;
- Orange – variability information;
- Purple – adaptation rules;
- Red/pink – dependencies, formulated as constraints;
- Yellow – context information;
- Blue – expressions.

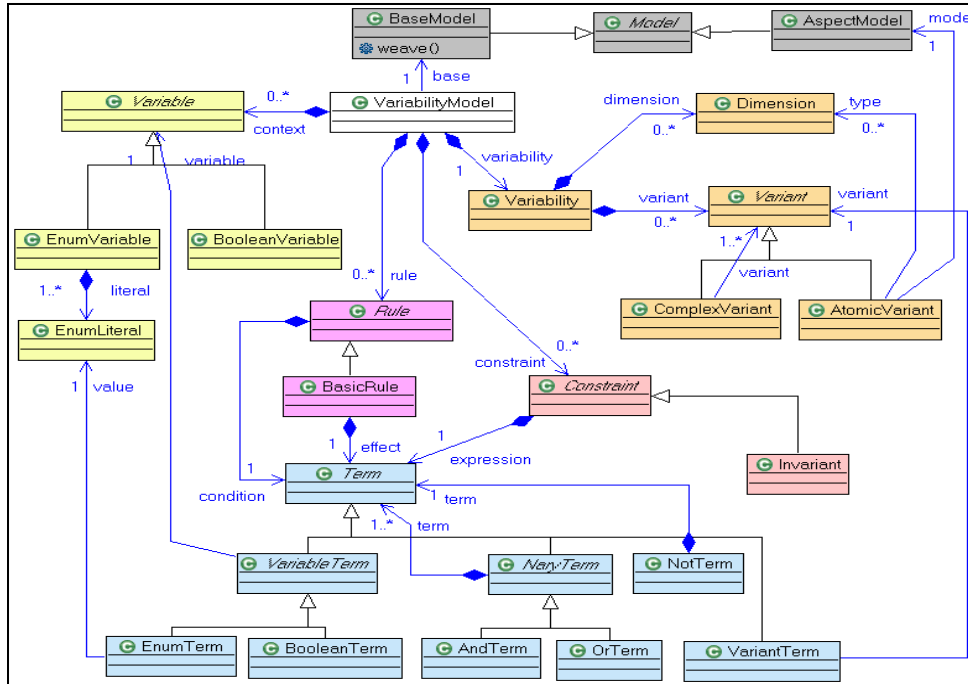


Fig. 2. Meta-model for variability and adaptation

This is to be considered a first version of our meta-model that has been created at an early stage in the DiVA project. It was created based on a set of simple examples such as the SDA described in this paper. During the project, the meta-model will evolve based on feedback and experiences with applying it to larger and more complex case studies. Nevertheless, at this point the meta-model is able to support modeling, simulation and validation activities. The following shows how the meta-model is instantiated for the SDA. To make the example readable we use a textual concrete syntax. This concrete syntax is processed by our prototype tool in order to build the adaptation model.

3.2 Modeling variability

Figure 3 shows a model of the variability information in our service discovery example, located in the section identified by the `#variability` keyword. We start by defining two variability dimensions; one for functional variability and another for different network protocols that the application can use. A variability dimension can best be described as a category of variants, while a variant is an aspect or concern that

is described outside of the base model and may vary to produce adaptation. So far, we have specialized variants further into atomic variants and complex variants. The latter is used to express a collection of several variants, thus forming a partial or full configuration. This concept was added because we encountered in our example that some combinations of variants were already foreseen during the requirements phase. As an example, the Discovery Agent functionality corresponds to having both the User Agent and the Service Agent functionalities. DA is thus defined as a complex variant referring to UA and SA.

```
#variability /* Variability of the application */
dimension Functionality : UA, SA
variant DA : UA, SA

dimension DiscoveryProtocol : ALLIA, SLP
```

Fig. 3. Variability in the Service Discovery Application

3.3 Modeling the context

Information about the context and sensors are delimited by the *#context* keyword. Currently, the meta-model supports two types of context variables: Booleans and enumerations.

The context model, as shown in Figure 4, starts with defining a variable for whether or not the device is running low on battery and, similarly, if the application has been elected as a Discovery Agent. Next, we have defined an enumeration that holds different roles. The application has to act as one of these roles at all time. Finally, there are two variables that tell which protocols are required, which can be one or many.

```
#context /* Context of the system */

boolean LowBatt // Battery is low
// Node has been elected Discovery Agent
boolean ElectedDA

// Node is required to act either as
// User Agent or as Service Agent
enum SrvReq : UA, SA

// Node is require to use one or
// more of the following prototcols
boolean ALLIAReq
boolean SLPReq
```

Fig. 4. Context of the Service Discovery Application

3.4 Modeling adaptation

Once the variability and context have been modeled, the adaptation rules can be specified. The adaptation rules link the context variables and the variants in order to specify the configuration to use with respect to a particular context. Currently, adaptation is based on simple *condition-action* rules. The *condition* part is a Boolean

expression based on the context information, while the *action* is a change in the configuration of variants.

```

/* Adaptation rules for functionalities */

rule BecomeDA : // Becomes a DA
  condition ElectedDA and not LowBatt and not DA
  effect DA

rule StopDA : // Stop being a DA
  condition (LowBatt or not ElectedDA) and DA
  effect not DA

rule BecomeUA : // Become a User Agent
  condition SrvReq=UA and not UA
  effect UA and not SA

rule BecomeSA : // Become a Service Agent
  condition SrvReq=SA and not SA
  effect not UA and SA

```

Fig. 5. Adaptation rules for the functionalities of the SDA

Figure 5 depicts the adaptation rules for the variants in the functionality category. The first rule is called “*BecomeDA*”, which is triggered when an application is elected as a discovery agent. If the device also has sufficient batteries and it is not a discovery agent already, the adaptation will proceed and the application will assume the role of a discovery agent.

3.5 Modeling constraints

Finally, Figure 7 shows the dependencies. These are currently modeled as constraints, more specifically invariants. For example, the first invariant states that the application must use at least one functionality variant. If it does not, an error message will be produced by the tool.

```

invariant AtLeastOneFunctionality : UA or SA
invariant NotDAWithLowBatt : not (LowBatt and DA)
invariant AtLeastOneProtocol : ALLIA or SLP
invariant NoSLPWithLowBatt : not (SLP and LowBatt)

```

Fig. 6. Invariants of the SDA

4 Simulation and Validation

The main benefit of using a model to describe adaptation is that it enables to process this model at design time in order to validate it [9]]. Based on the meta-model defined in the previous section we have defined a simulator and automated the verification of invariants. This section describes the way the simulator is built and how it allows checking for termination of adaptation rules and verification of invariant properties.

4.1 Simulation model and Implementation

The goal of the simulation is to build a model of the potential configurations and adaptations of the application. To do that, the simulation starts from an initial configuration and applies the adaptation rules to move to a new configuration. Figure X presents the simulation model. According to this model, a simulation is composed

of a set of configurations and a set of adaptations between these configurations. Each configuration refers to a set of variants and a set of variable terms. The variants correspond to the aspect to be woven in order to build this configuration [7]. The Variable terms define the state of the context variables for this configuration. An adaptation links a source configuration with a target configuration. An adaptation is triggered by a context event and refers to one or more adaptation rules. The context event is a change in the values of one or more context variables.

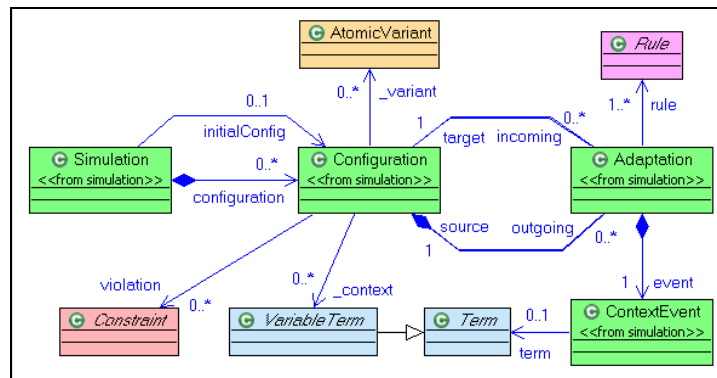


Fig. 7. Simulation model

Based of this simulation model, a prototype simulator has been implemented using the Kermeta platform [2]. The simulator starts from an initial configuration and for each variation of the context variables it evaluates the guards of the adaptation rules. If the guard of an adaptation rule is true in the new context then this rule must be applied and the guards of all the rules are evaluated again. Adaptation rules are applied until none of their guards evaluated to true.

4.2 Simulation output

The output of a simulation can be rendered as a graph in which each node is a configuration and each edge is an adaptation. **Fig. 8** shows an excerpt of the simulation graph for the service discovery application. The complete simulation graph for this example contains 24 configurations obtained by aspect weaving and 70 adaptations. In the label of each node, the first line corresponds to the values of the context variables and the second line to the set of aspects that should be used to create the corresponding configuration. Each edge in the graph corresponds to an adaptation to a change of one context variable. The label of the edges starts with the context variable change and details the set of adaptation rules that were applied. In the graph presented **Fig. 8** the configurations have been colored in order to visualize easily the battery level. Configurations for which the battery is high are displayed in green and configurations with low battery are displayed in orange.

4.3 Constraint checking and rule termination

The main benefit of the simulation model is to allow for validating the adaptation rules at design time. As shown in the previous section the adaptation graph can be

visualized and colors can be used in order to highlight specific properties. This allows for a manual validation of the specified rules. In addition, the simulation process can identify live-locks and dead-locks in the adaptation graph and allows to automatically verifying invariants on the system.

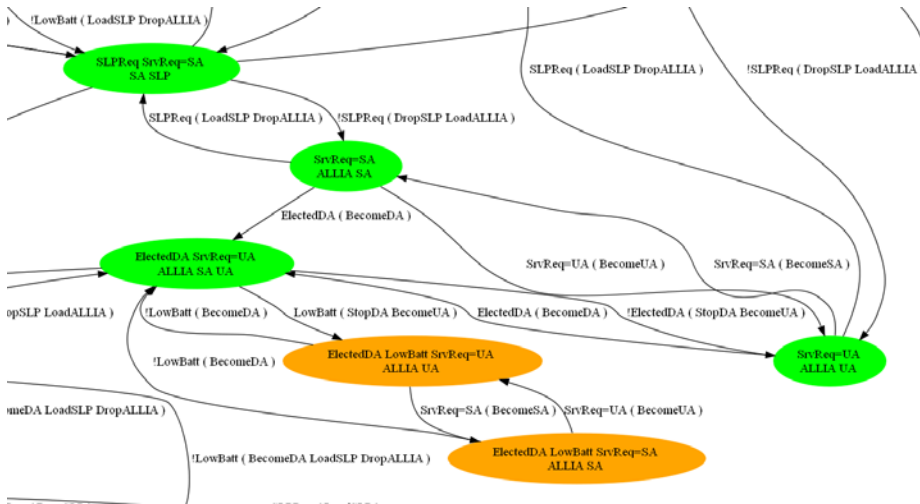


Fig. 8. Excerpt of the simulation graph for the SDA

Dead-locks in the simulation graph correspond to cases where some adaptation rules lead to a configuration from which the system cannot adapt. In a design, this could be done voluntarily but in most cases this is due to some incorrect or missing adaptation rules. Live-locks correspond to cases where the system bounces between several configurations while the context is not changing. This situation always reveals an error in the adaptation rules. The simulator can identify live-locks while it compute the simulation graph. For a single change of the context, no adaptation rule should be able to apply twice. Indeed, if after applying a rule (and possibly some others), the same rule can apply again then the rule could be applied an indefinite number of times. When this situation is detected by the simulator, it reports an error in the rules and provides the configuration in which the problem occurs and the sequence of rules which is looping.

The meta-model presented in section 3 allows defining invariants on the system. These invariants are checked by the simulator on all the configurations that are created during the simulation. Any violation of these invariants reveals an error in the adaptation model.

5 Adapting the system at runtime

In this section, we present how we actually adapt a running system using the rules we presented in Section 3. In order to trigger the rules, we need to monitor the state of the system itself and the execution context e.g., memory, CPU usage, available network

bandwidth, battery level, etc. For this purpose we intend to reuse the *Intel Mobile Platform Software Development Kit* [1] that already offers a large set of probes. This SDK is freely available and provides a Java API implementing these probes. Using these probes, we have to implement the variables related to the execution context e.g., *lowBatt*. For example, we can specify that:

```
lowBatt = batteryInstance.percentRemaining < 15
```

However, defining the variable *lowBatt* in this way may be too strict. For example, if the battery level goes under 15%, the system will adapt. But, if the user plugs the system to power supply, the battery level will rapidly increase and the system may adapt again because the battery is not low anymore. In this case, the system adapts twice whereas it would have been preferable to do nothing as the adaptation process may be time consuming.

In order to tackle the instability of rules, we will use WildCAT 2.0, currently still under development. WildCAT [4] is an extensible Java framework that eases the creation of context-aware applications. It provides a simple but yet powerful dynamic model to represent the execution context of a system. The context information can be accessed by two complimentary interfaces: synchronous requests (pull mode: application makes a query on the context) and asynchronous notifications (push mode: context raises information to the application). Internally, it is a framework designed to facilitate the acquisition and the aggregation of contextual data and to create reusable ontologies to represent aspects of the execution context relevant to many applications. A given application can mix different implementations for different aspects of its context while only depending on WildCAT simple and unified API. The version 2.0 of WildCAT allows defining SQL-like requests on the environment model and integrate the notion of time. For example, it is possible to trigger a rule when the battery has been lower than 15% for more than 3 minutes.

When a rule is triggered, the associated variants become active. In other words, we weave the aspects associated to each variant in the base model. Aspect weaving is currently performed with SmartAdapters [6]. Then, we compare the woven model with the reference model, obtained by introspection over the running system. This comparison generates a diff and match model specifying what has changed in the woven model. By analyzing this model, we automatically generate a safe reconfiguration script that is then applied to the running system. Aspect weaving and automatic adaptation are described in more details in [6, 8].

6 Discussion and Conclusion

This paper presents our ongoing work on modeling adaptation. So far, based on the meta-model we have modeled, simulated and checked a few toy adaptive applications. However we have also identified the need for more expressiveness in order to describe the variants, the context and the adaptation rules. Our objective is to build on top of the current meta-model in order to identify a restricted set of concepts relevant to the modeling of variability and adaptation. At this stage, we have identified two specific issues.

Firstly, in their current form, the number of adaptation rules can quickly grow as the number of context elements and variants increase. Our main goal is to tackle the problem of an explosive growth in the number of configurations and the artifact to be

used in their construction. However, we do not want to move the complexity associated into the rules as a consequence. Consequently, as a step towards improving our adaptation rules, we aim to express the *rules* using semantics. In that sense, the rule should be on the form “*choose a set of variants with properties that match the current context*”. The above embraces a more declarative approach. Although, sometimes we still might want to allow rules on variant configurations since pre-defined full or partial configurations might be extracted or derived from the requirements straightforwardly, as was the case in our variability model.

Secondly, our current simulation prototype enumerates all the configurations and adaptations between them. While this is very useful and works well while the number of configurations is manageable, this approach has the typical model-checking scalability issues when the number of configuration and adaptation grows. Several techniques can be combined in order to keep the simulation space manageable, for example, adding constraints on the context, considering sub-sets of variability dimensions or using heuristics to limit the depth of simulations. In the context of the DiVA project, we plan to experiment with industrial adaptive applications in order to choose the most appropriate solutions to this scalability issue.

For the runtime, as future work we plan to automate as much as possible the implementation of the triggers. For example, it is easy to quantify the domain in which a battery evolves: 0 to 100. But, defining what a low level for a battery is may be more difficult. We previously said that a battery is low if the remaining percentage is lower than 15 for 3 minutes. However, this kind of information is generally not specified in requirement documents and developers have to infer the information from their knowledge and/or based on experimentation. We plan to use Fuzzy logic to help in defining and implementing triggers. Providing a global domain (0 to 100) and some qualifiers (“high”, “medium”, “low”), the fuzzy logic can determine, for a given observed value (e.g., 17%) if the battery is “low”, “medium”, etc. Fuzzy logic can help us in filling the gap between requirement (qualitative descriptions) and implementation (quantitative observations) and allows keeping high-level adaptation rules at runtime.

References

- [1] <http://ossmpsdk.intel.com/>.
- [2] <http://www.kermeta.org/>.
- [3] Nelly Bencomo, Robert France, and Gordon Blair. 2nd international workshop on models@run.time. In Holger Giese, editor, *Workshops and Symposia at MODELS 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007.
- [4] P.C. David and T. Ledoux. WildCAT: a generic framework for context-aware applications. In *MPAC'05: 3rd Int. Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 1–7, New York, NY, USA, 2005. ACM.
- [5] 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.
- [6] Philippe Lahire, Brice Morin, Vanwormhoudt Gilles, Alban Gaignard, and Jean-Marc Jézéquel. Introducing variability into aspect-oriented modeling approaches. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 07)*, USA, October 2008.

- [7] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jezequel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MODELS'08 Conference*, France, 2008.
- [8] Brice Morin, Gilles Vanwormhoudt, Philippe Lahire, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Managing variability complexity in aspect-oriented modeling. In *Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08)*, Toulouse, France, October 2008.
- [9] Ji Zhang and Betty H.C. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering (ICSE'06)*, China, 2006.