Software Evolution Towards Model-Centric Runtime Adaptivity

Mehdi Amoui
University of Waterloo
Waterloo, Canada
mamouika@uwaterloo.ca

Mahdi Derakhshanmanesh
University of Koblenz-Landau
Koblenz, Germany
manesh@uni-koblenz.de

Jürgen Ebert
University of Koblenz-Landau
Koblenz, Germany
ebert@uni-koblenz.de

Ladan Tahvildari
University of Waterloo
Waterloo, Canada
ltahvild@uwaterloo.ca

***Abstract*—Runtime adaptivity is a promising direction towards achieving adaptive behavior for software systems that operate within highly dynamic and non-deterministic environments. Model-centric approaches have proven to be able to successfully address various aspects of runtime adaptivity. In this paper, we propose a target architecture for self-adaptive software systems and show how it facilitates adaptation by interpreting models at runtime. Our approach supports adaptivity using models, which are causally connected to the software application. These models can be queried and transformed dynamically in reaction to changes in the software system's operating environment. We demonstrate how to implement an infrastructure to support the target architecture, and how to prepare and integrate non-adaptive software to comply with this architecture.**

***Keywords*-software evolution; runtime adaptivity; self-adaptive software; model transformation; models at runtime**

I. Introduction

A Self-Adaptive Software System (SASS) is a self-conscious reflective system that is able to modify its own behavior in response to changes in its operating environment [1]. It is also called an autonomic / self-managing system [2]. An SASS follows a reference architecture that comprises two main subsystems: the adaptation manager (autonomic manager) and the adaptable software (managed element) [3]. The adaptation manager acts as an external controller that observes the adaptable software for changes in its operating states and selects appropriate adaptive behavior accordingly. These two subsystems are connected through sensor and effector interfaces, which shapes a closed feedback control loop [3].

Due to the increasing demand for adaptive systems and the high costs of developing them from scratch, reengineering current systems into adaptive ones is inevitable. Several attempts have been made to renovate and retrofit legacy systems into adaptive systems (e.g [4], and [5]). Such a migration involves activities that are well known in the field of software evolution and maintenance, such as tasks related to reverse engineering, program comprehension, impact analysis, or change propagation. However, evolution towards an SASS also involves other less familiar challenges, such as: (i) The migrated SASS must preserve its default behavior if no adaptation is performed at runtime. (ii) Any types of changes to the original software, made for the purpose of introducing adaptivity properties, must be kept as separate as possible. (iii) Locating software application fragments that may be involved in runtime adaptation steps requires a good knowledge of the software system and the adaptation requirements. (iv) Changes to be made to the current software must be kept minimal and cost effective.

We tackle the challenges above by proposing a new target architecture for SASSs, and combine it with an evolution process. This architecture extends the reference architecture of an SASS [3] and facilitates the evolution of legacy software towards adaptivity by supporting automatic generation, interpretation, and maintenance of runtime models.

II. Target Architecture

In this section, we introduce a blueprint for a model-centric SASS as a target architecture for our evolution process towards runtime adaptivity. The architecture is mainly organized into layers as depicted in Figure 1. It consists of two detachable subsystems, which are causally connected to each other: the *adaptable software* and the *adaptation framework*.

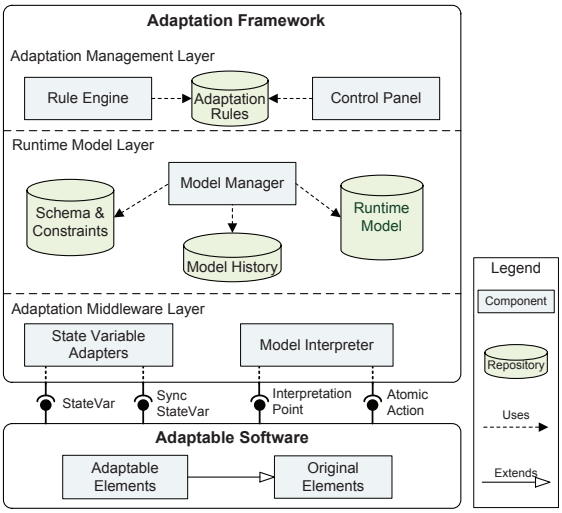

Figure 1. The Target Architecture of a Model-Centric SASS.

A. Adaptable Software

The *adaptable software* is an evolved version of the original software, which represents the managed subsystem of the SASS. It preserves the behavior of the original application, and its program elements (such as classes, methods, attributes, etc.) extend those of the original software. However, the elements may be changed during the evolution process to support adaptivity.

Those elements of software that are in need for adaptation, or can provide information about changes in the application's operating environment, must be exposed to the adaptation framework. This exposure can be achieved by instrumenting the elements to expose a set of interfaces. The adaptation framework accepts four interfaces to be provided by the adaptable software, as shown in Table I.

Table I
INTERFACES PROVIDED BY THE ADAPTABLE SOFTWARE.

Interface	Description
StateVar	exposes state variables that hold information about the operating environment. Changes in the value of these variables is reified to the runtime model.
SyncStateVar	exposes state variables that can be changed in the runtime model as well. New values are propagated back to the adaptable software.
Interpretation Point	exposes a handle to positions in the adaptable software's control flow, where an associated part of the behavioral model shall be executed.
AtomicAction	exposes methods that hold either default or alternate behavior. These actions can be composed in the behavioral model.

B. Adaptation Framework

The adaptation framework is the counterpart of the *Adaptation Manager* subsystem of the reference architecture for SASSs described in [3]. It has a layered architecture as follows:

1) Adaptation Middleware Layer: This layer is responsible for all possible ways of interaction between the adaptable software and the framework. State transitions in the adaptable software are propagated to the runtime model via this layer. It also ensures that changes in the runtime model affect the adaptable software. The components of this layer are:

- *State Variable Adapters:* handle the propagation of changed values of instrumented state variables to the runtime model.
- *Model Interpreter:* executes method compositions, as configured in behavioral sub-models.

2) Runtime Model Layer: The runtime model layer models parts of the adaptable software and contains utility components for model management. The main components of this layer are:

- *Runtime Model:* contains a state and behavioral model of the adaptable software. The *state model* describes elements of the adaptable software that represent its state via their current runtime values. The *behavioral model* contains sub-models that orchestrate *atomic actions* (methods), exposed by the adaptable software, to be executed by the model interpreter at predefined points in control flow. The runtime model conforms to a schema.
- *Schema:* describes the abstract syntax of the language to express runtime models and defines constraints on its elements. For instance, behavioral models can be expressed using different languages such as petri nets, activity diagrams, state charts, or even decision trees.
- *Model Manager:* keeps the runtime model clean and in sync with the adaptable software. It acts as a controller for all accesses to the runtime model. It is also responsible for the evaluation of constraints when the runtime model changes.
- *Model History:* is the storage of temporal snapshots of the runtime model as additional decision support data. The adaptation management layer can access this repository (by querying it) via the model manager.

3) Adaptation Management Layer: This layer senses changes in the runtime model and takes actions to adjust it. It is composed of the following components:

- *Adaptation Rules:* describe the adaptation logic. Each rule is composed of three parts: (i) the *event* representing a change in the runtime model that requires the rule engine's reaction; (ii) the *condition* that is the boolean evaluation of a query on the runtime model, which tests whether a rule should be applied; and (iii) the *action* that is essentially a model transformation.
- *Rule Engine:* receives change events from the model manager and runs queries via the model manager to retrieve detailed information on the current or past state of the system. It is sensitive to different types of (change) events. Based on the adaptation rules that are specific to the received event, the rule engine plans necessary adjustments. The plan is expressed as a (compound) model transformation to be executed on the runtime model, involving the model manager.
- *Control Panel:* allows administrators to query and view parts of the runtime model, investigate the model history, and modify adaptation rules.

III. IMPLEMENTATION

In this section, we mainly describe one possible way for realizing the adaptation framework of Figure 1, which we used in our prototypical implementation.

A. Runtime Modeling with the TGraph Approach

The runtime model and its operations are based on graph technology. Models, schemas as well as queries and transformations are implemented in the technological space of TGraphs [6]. Furthermore, we make use of JGraLab[1], a versatile class library that can create Java source code based on meta-models (schemas). The generated classes can then be used in custom applications to utilize TGraphs as data structure. In addition, JGraLab can create, query and transform TGraphs at runtime. Finally, the runtime model can be queried conveniently using the *Graph Repository Query Language* (GReQL) [7].

We describe the runtime model schema with UML class diagrams. An excerpt of our currently implemented schema is illustrated in Figure 2. Accordingly, every runtime model is composed of a set of `StateVariable` and `Activity` vertices. Moreover, behavioral models are expressed as variants of UML activity diagrams, and state variables are used to store exposed data from the adaptable software.

[1] http://www.ohloh.net/p/jgralab

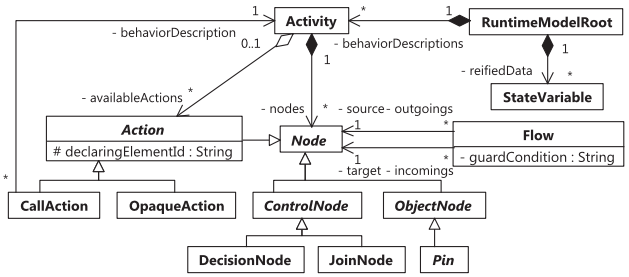

Figure 2. Excerpt of the Runtime Model Schema.

B. Connecting the Framework using AOP

We glue the adaptable software to the adaptation framework by using JBoss AOP[2], an aspect oriented framework written in Java. Using bytecode manipulation, the necessary code for value propagation of state variables from the adaptable software application to the runtime model is injected. Similarly, the code for model interpretation is injected at previously annotated points in the adaptable software's control flow.

C. Model Interpretation and Reflection

The model interpreter walks along paths of an activity which describe (alternate) behavior to be executed at a previously annotated interpretation point. Each `Action` vertex of an activity in the runtime model has a trace link (e.g. the qualified method name) to a method in the adaptable software. The model interpreter resolves this meta-data to invoke these methods using Java reflection. If the associated behavioral sub-model is not tagged to be interpreted, for instance because it describes the same behavior as the adaptable software's default implementation, then no interpretation takes place and the adaptable software's default implementation is invoked.

IV. Evolution Process

To evolve legacy software to our proposed target architecture, we introduce a high-level process model as depicted in Figure 3. This evolution process requires a set of adaptation requirements, which specifies adaptive behavior of a SASS. The output of the process is a SASS which satisfies the adaptation requirements.

A. Pointcut Specification

The evolution process starts by analyzing adaptation requirements and specifying the program elements that should be provided by the adaptable software via its interfaces described in Table I to communicate with the adaptation framework. We name these required elements *pointcuts*. At this stage, we only determine and describe the pointcuts, regardless of the presence of the program elements to support them. In the next step, we analyze the original software to figure out potential program elements that can be matched with these pointcuts.

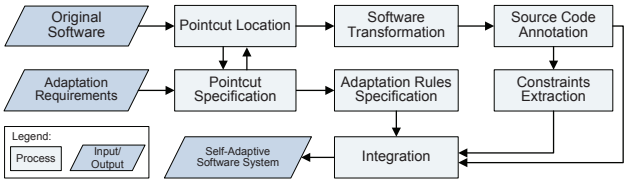

Figure 3. Overview of the Evolution Process Towards an SASS.

B. Pointcut Location

After we specify the pointcuts, we locate the relevant position of the pointcuts in software by mapping existing software elements to the interfaces. The supporting elements might not exist in original software. Even existing elements may require modification before we can annotate them as a pointcut.

A developer analyzes original software with regards to the list of potential pointcuts prepared in Subsection IV-A. The list may also include those pointcuts that are required but cannot be provided by software (e.g. due to high costs of change to include supporting elements).

We label each pointcut as either *available*, *missing* or *located*, based on how current software elements support them. *Available* pointcuts are those, that can be assigned to existing program elements. *Located* ones can be added by just modifying current program elements, and *missing* pointcuts need additional elements to support them.

C. Software Transformation

In order to expose missing and located pointcuts, we must transform the original software. The transformations are either in the form of adding new program elements, modifying existing ones, and annotating them.

1) Adding Missing Elements: We add and initialize missing state variables. The alternate actions are also added to the same classes that contain their corresponding default actions. Any modification to other elements for accessing new elements must be done through the *Modifying Existing Elements* step for preserving the default behavior.

2) Modifying Existing Elements: The modifications of already existing elements is another step towards adding pointcuts that enable behavioral variability. Variability in runtime behavior is generally achieved by either *parameter adaptation* or *compositional adaptation* [8].

Figure 4 illustrates two approaches to achieve adaptivity at runtime. Figure 4(a) represents the flow graph of a non-adaptable software application with a missing *Alternate Action* to replace the default behavior provided by the available *Default Action* at node 2. After adding the missing *Alternate Action*. We can swap the behavior by either parameter or compositional adaptation.

For *parameter adaptation*, the application flow can be redirected to an *Alternate Action* at a *Turning Point*. A turning point is a branch in the control flow with a well-defined decision criterion that is realized by a `SyncStateVar`. In Figure 4(b), the same graph is augmented by an alternate action at the turning point, node

[2]http://www.jboss.org/jbossaop/

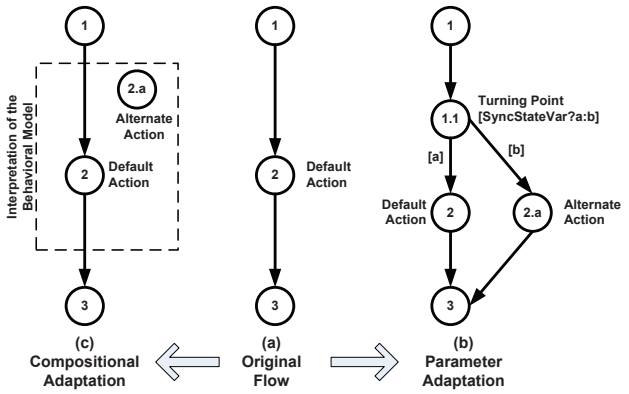

Figure 4. Modifications Towards Runtime Adaptation.

1.1. The `SyncStateVar` of node 1.1 can be tuned by the adaptation framework to select either the *Default Action*, node 2, or the new *Alternate Action*, node 2.a.

For *compositional adaptation*, the adaptation framework replaces a block of the adaptable software's control flow with the execution of modeled behavior at runtime. Figure 4(c) illustrates how this change takes place. The rule engine is able to replace node 2 with 2.a in the behavior runtime model. Node 2.a has to satisfy the pre-conditions and post-conditions of node 2. We extract an *interpretation* method, and introduce it as a pointcut. This pointcut is bound to an activity in the behavioral model which describes the actions to be executed using the available atomic actions. At runtime, model interpretation takes place when the control flow reaches the *interpretation point*.

D. Software Annotation

Finally, we annotate all the prepared elements. Each source code annotation is used for load-time instrumentation according to an AOP pointcut to inject state variable adapters or the model interpreter hooks into Java classes.

E. Adaptation Rules Specification

Based on the adaptation requirements and the pointcuts, this step involves writing concrete adaptation rules. For each rule, we select an appropriate subset of exposed elements from each interface. The next step is encoding the conditions as queries on the state variables, and encoding actions as transformations of the runtime model.

F. Integration

This step integrates the components that are developed in the previous steps to finally create an SASS. Involved steps are the generation of a default runtime model by utilizing the source code annotations, as well as the initialization of the adaptation framework using the provided repository with adaptation rules. The *Integration* process consists of two phases: (i) *startup configuration* and (ii) *load-time initialization*.

At startup, an instance of the adaptable software is initialized, together with an instance of the framework and configuration files for JBoss AOP. At load-time, hooks to state variable adapters and to the model interpreter are injected into compiled and annotated Java classes of the adaptable software to connect it to the adaptation framework. Furthermore, a default instance of the runtime model is generated, which includes representations for the annotated state variables, as well as default behavioral submodels for all interpretation points.

V. Concluding Remarks

In this paper, we presented a new model-centric architecture for SASSs and an evolution process which supports the systematic migration of the original software towards this proposed architecture. Query, transform and interpretation services on the runtime model are at the heart of this approach.

At present, our implementation of the adaptation framework lacks some of the proposed features, such as the model history and the external control panel. However, our early experiments using this prototype implementation exhibit, that compositional adaptation is a more flexible way of implementing adaptivity, as the system's behavior can be controlled by the runtime model's behavioral model. This approach adds some overhead, though.

In contrast, parameter adaptation seems more suitable in cases where highly dynamic changes at runtime are not required. This approach has less overhead on system performance, as a main part of adaptation is done during the evolution process at source code level, and no model interpretation is involved at runtime.

References

[1] P. Oreizy *et. al.*, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[3] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] J. Parekh, G. Kaiser, P. Gross, and G. Valetto, "Retrofitting autonomic capabilities onto legacy systems," *Cluster Computing*, vol. 9, no. 2, pp. 141–159, 2006.

[5] J. Zhang and B. H. C. Cheng, "Towards re-engineering legacy systems for assured dynamic adaptation," in *Proceedings of the International Workshop on Modeling in Software Engineering*, 2007, p. 10.

[6] J. Ebert, V. Riediger, and A. Winter, "Graph Technology in Reverse Engineering, the TGraph Approach," in *10th Workshop Software Reengineering*, ser. GI Lecture Notes in Informatics, vol. 126. Bonn: GI, 2008, pp. 67–81.

[7] J. Ebert and D. Bildhauer, "Reverse engineering using graph queries," in *Graph Transformations and Model-Driven Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 5765, pp. 335–362.

[8] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing adaptive software," *Computer*, vol. 37, pp. 56–64, 2004.