3rd Workshop on Models@run.time

at MODELS 2008

(Proceedings)

Edited by

Nelly Bencomo
Gordon Blair
Lancaster University
Robert France
Colorado State University
Freddy Muñoz
INRIA, France
Cedric Jeanneret
University of Zurich, Switzerland

Technical Report COMP COMP-005-2008
Lancaster University

LANCASTER
UNIVERSITY

3rd Workshop on Models@run.time at MODELS 2008
30 September, 2008, Toulouse, France

Program Committee

Betty Cheng
Michigan State University, USA
Fabio M. Costa
Federal University of Goias, Brazil
Anthony Finkelstein
UCL, UK
Jeff Gray
UAB, USA
Oystein Haugen
SINTEF, Norway
Jozef Hooman
ESI, The Netherlands
Gang Huang
Peking University, China
Paola Inverardi
University of L'Aquila, Italy
P.F.Linington
University of Kent, UK
Jean-Marc Jezequel
Triskell Team,IRISA, France
Rui Silva Moreira
UFP, INESC Porto, Portugal
Andrey Nechypurenko
Siemens, Germany
Oscar Nierstrasz, Switzerland
University of Berne
Eugenio Scalise
UCV, Venezuela
Arnor Solberg
SINTEF, Norway
Thaís Vasconcelos Batista
UFRN, Brazil
Steffen Zschaler
T.U. Dresden, Germany

Organizing Committee

Nelly Bencomo, Gordon Blair, Lancaster University, UK
Robert France, Colorado State University, USA
Freddy Muñoz, INRIA, France
Cedric Jeanneret, University of Zurich, Switzerland

Preface

Welcome to the 3rd Workshop on Models@run.time at MODELS 2008

This document contains the proceedings of the 3rd Workshop on Models@run.time that was co-located with the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS). The workshop took place in the beautiful city of Toulouse, France, on the 30th of October, 2008. The workshop was organized by Nelly Bencomo, Robert France, Gordon Blair, Freddy Muñoz, and Cédric Jeanneret. From a total of 20 papers submitted 6 full papers, 6 short papers, and a 1 demo were accepted. This volume gathers together all the 6 full papers accepted at Models@run.time 08.

We would like to thank a number of people who contributed to this event, especially the members of the program committee who acted as anonymous reviewers and provided valuable feedback to the authors. We also thank to the authors of all submitted papers are thanked for helping us making this workshop possible.

November, 2008

Nelly Bencomo,
Gordon Blair
Lancaster University, UK
Robert France
Colorado State University, USA
Freddy Muñoz
INRIA, France
Cedric Jeanneret
University of Zurich, Switzerland

CONTENTS

Long Papers

Embedding State Machine Models in Object-Oriented Source Code, Michael Striewe, Moritz Balz and Michael Goedicke …………………………………………6

Model-Based Traces, Shahar Maoz……………….……………………….………..16

Mutual Dynamic Adaptation of Models and Service Enactment in ALIVE, Athanasios Staikopoulos, Sebastien Saudrais, Siobhan Clarke, Julian Padget, Owen Cliffe and Marina De Vos…………….…………………………………….………..26

Modeling and Validating Dynamic Adaptation, Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, Jean-Marc Jézéquel………….…………..………..36

Runtime Models for Self-Adaptation in the Ambient Assisted Living Domain, Daniel Schneider and Martin Becker…………….…………………………………..47

FAME---A Polyglot Library for Metamodeling at Runtime, Adrian Kuhn and Toon Verwaest…………………………………………………………….…………..57

Short papers

A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service, Audrey Occello, Anne-Marie Dery-Pinna and Michel Riveill…………….………………………………………………………………..67

Runtime Models to Support User-Centric Communication, Yingbo Wang, Peter J. Clarke, Yali Wu, Andrew Allen and Yi Deng…………….……………………..…..77

A Framework for bridging the gap between design and runtime debugging of component-based applications, Guillaume Waignier, Prawee Sriplakich, Anne-Francoise Le Meur and Laurence Duchien. …………….…………………….……..87

A Model-Driven Approach for Developing Self-Adaptive Pervasive Systems, Carlos

Cetina, Pau Giner, Joan Fons and Vicente Pelechano. ……………..………………..97

An Execution Platform for Extensible Runtime Models, Mario Sanchez, Ivan Barrero, Jorge Villalobos and Dirk Deridder…………..…………………………107

Model-driven Management of Complex Systems, Brian Pickering, Sylvain Robert, Stephane Menoret and Erhan Mengusoglu……………..……………………..…….117

Demo

K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines, Brice Morin, Olivier Barais and Jean-Marc Jézéquel……………..127

Embedding State Machine Models in Object-Oriented Source Code

Moritz Balz, Michael Striewe, and Michael Goedicke

University of Duisburg-Essen
{moritz.balz,michael.striewe,michael.goedicke}@s3.uni-due.de

Abstract. This contribution presents an approach to maintain state machine model semantics in object-oriented structures. A framework is created that reads and executes these structures at run time and is completely aware of the model semantics. The goal is to embed such structures in arbitrary large systems and delegate program control to the framework. Hence we can debug and validate the system at run time and apply monitoring with respect to state machine model characteristics.

1 Introduction

State machines can be comprehensively specified, simulated and verified at design time. We present an approach to retain model semantics in executable systems to allow debugging, validation and monitoring at run time. Our approach will be introduced by a real-world example and formalized later on. The example depicts a load generator application in which we implemented a state machine model that controls program execution and invokes existing business logic.

Traditional ways to translate models into source code by either manual implementation or automated code generation [1] are not suitable for this application: The inherent loss of semantic information entails that models are related to developed systems only by the developer's knowledge [2], thus preventing automatic back tracking of changes [3]. Model Round-Trip Engineering concepts [4] make code synchronisation possible but require manual effort and are thus error-prone [5]. Additionally, generation tools often lack the capabilities to integrate their output into existing systems like our load generation application. Even if only one modeling language is used, the need to regenerate parts of the source code after local changes contradicts a gradual integration [6].

Model execution engines (e.g. Executable UML [7]) can avoid the mentioned problems by interpreting model descriptions. This is not appropriate either, when the system can not be entirely defined in an executable model. This leads to a loss of type information at integration layers between model and residual source code. In addition, bad performance might be experienced due to heavyweight integration layers or necessary data conversion.

Common to these approaches is the permanent existence of different types of model representations at several development stages or parts of the run time

system. Our approach aims at avoiding these differences by storing state machine model semantics explicitly in object-oriented structures. The goal is to embed such structures in arbitrary large systems and delegate program control to the framework. The framework analyzes the code structures and extracts an internal run time representation of the state machine. It walks through the state machine by evaluating guards and updates and invokes methods that represent transitions accordingly. These methods contain arbitrary code and connect the state machine to the application logic. Our approach naturally ensures that the executed system is equivalent to the designed model. Moreover, we can debug and validate the system at run time and apply monitoring with respect to state machine model characteristics. This benefits come at the cost of having to obey rules while writing the related source code structures, but without the (often not realized) effort to maintain the source code and a separate model at the same time. Section 2 of this contribution demonstrates the basic ideas by example, while sections 3 and 4 explain the formal approach and its mapping to a Java implementation in detail. Sections 5 and 6 show related work and draw the conclusions.

2 Example

We illustrate our approach on the basis of the mentioned load generator application that has been developed using Java. The control mechanism is modelled as a state machine. The program flow starts with some preparations for the measurement. Then an actual measurement run is performed wherein load is generated by worker threads. The result is evaluated and the number of workers is increased and decreased in order to explore the load behaviour of a system under test. The last two steps are repeated until a measurement result is achieved. The states before and after the measurement have transitions that fire depending on the last measurement results. During transitions the application will e.g. increase or decrease workers.

In order to maintain these state machine semantics in the source code in addition to application logic, we create classes that represent states. Methods in these state types represent transitions that invoke business logic and are decorated with meta data referencing the transition target state. This leads to a network of state classes being connected by transition methods that represent the state machine in object-oriented structures, which is partly shown in figure 1.

The state machine starts at the initial state and performs some preparations with the first transitions. Then it performs an actual measurement and reaches a state named “AfterMeasurement” depicted at the top of figure 1. The implementation of this state is shown in listing 1.1 with minor omissions. It shows the way classes and methods are interpreted as states and transitions and how the actual application components are invoked.

State classes are simply marked with the `IState` interface. Transition methods are marked by an annotation that refers to the `target` state class and a `contract`

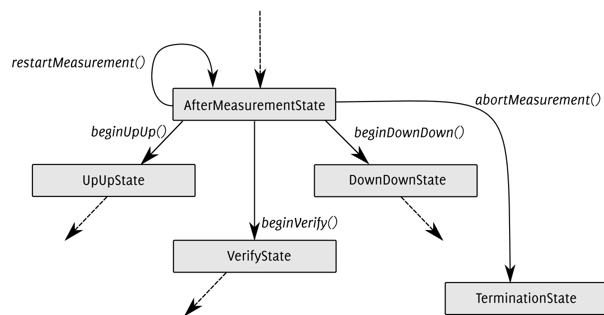

Fig. 1. State classes and transition methods. The node descriptions are class names, the edge labels represent method names.

```
public class AfterMeasurementState implements IState {
  @Transition(target = AfterMeasurementState.class, contract = RestartContract.class)
  public void restartMeasurement(MeasurementModule actor) {
    actor.increaseNumberOfRestarts();
    actor.doMeasure("Restarted␣measurement");
  }

  @Transition(target = UpUpState.class, contract = BeginUpUpContract.class)
  public void beginUpUp(MeasurementModule actor) {
    actor.resetRestarts();
    actor.beginUpUp();
    actor.doMeasure("Exploration␣by␣distance␣upwards");
  }

  // . . .

  @Transition(target = TerminationState.class, contract = AbortContract.class)
  public void abortMeasurement(MeasurementModule actor) {
    actor.terminateMeasurement();
  }
}
```

Listing 1.1. Class `AfterMeasurementState` with some outgoing transitions

```
public class BeginUpUpContract implements IContract<IMeasurementVariables> {
  public boolean checkCondition(IMeasurementVariables vars) {
    return (!vars.getAbort() && !vars.getRestart() && vars.getTooLow());
  }

  public boolean validate(IMeasurementVariables before, IMeasurementVariables after) {
    return (after.getNumberOfWorkers() == (before.getNumberOfWorkers() + before.
          getWorkerDistance()));
  }
}
```

Listing 1.2. Guards and updates in `BeginUpUpContract`

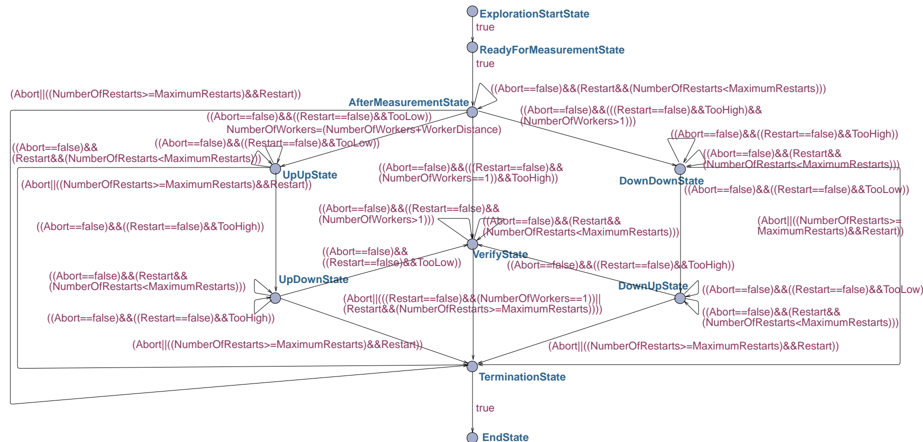

Fig. 2. The state machine model of the load generator

class containing guards and updates for this transition. The method contents use a facade object `actor` that encapsulates the application logic and separates it from the model structures.

Listing 1.2 shows the class containing guards and updates for the `BeginUpUp` transition. Variable values defining the state space are provided by another encapsulating type, denoted `vars`, and used to evaluate guards in the `checkCondition` method. State changes during transitions can be verified with the `validate` method, that does not perform the actual update, but checks whether it has taken place in the implementation as desired. Simple comparisons and logical operations in both methods are mapped one-to-one. Updates are represented as tests for equality as shown in listing 1.2, where the worker increment is validated by checking that $numberOfWorkers' = (numberOfWorkers + distance)$.

Embedding model semantics in code structures allows us to read the complete model at design time and validate it in state machine modeling tools. So far a complete extraction is possible for UPPAAL [8]: The state machine shown in figure 2 is completely extracted from the existing source code and just laid out manually. How to do this is discussed in section 3.4.

As can be seen, these structures are able to cover states, transitions, guards and updates and hence include nearly all state machine semantics. The only missing item is the initial state, which is covered by the execution component necessary to walk through the machine. This will be discussed in section 3.3, after the formalization of our approach.

3 Formalization of the Approach

The code structures containing the state machine model semantics will be executed at run time and used as an input to state machine modelling tools

at development time. Thus we need a universal definition as a formal base for well-defined interpretations. For our approach we define state machines as $M = \{S, T, V, P, U\}$ with

- S a set of states.
- $T \subset S \times S$ a set of transitions between states.
- $V = \{v_1 \ldots v_n\}$ a set of named variables.
- $P = \{p_t | t \in T\}$ a set of guards for each transition.
- $U = \{u_t | t \in T\}$ a set of updates for each transition.

The application state is modified only when transitions are fired. Execution control will be passed to application components at this point of time and return to the state machine when the next state is reached. The variables are used at modeling time for state space analysis and are provided at run time by a source code component representing the application state.

Each guard is an expression related to one or more variables that evaluates to a boolean value. Guards will be used at design and run time to decide which transition in the current state should fire. Comparisons and basic arithmetic operations can be performed on variables and literals inside expressions.

Each variable update consists of atomic assignments that define either a single value or a range of values as update for one variable. New values may be constants or variable values which can be connected using basic arithmetic operations as above. Additionally, each variable value of the previous state is accessible to allow relative changes. At design time updates are used to define and change the model state space. At run time they can be interpreted as post-conditions in order to monitor if the application is in an expected state.

3.1 Embedded Model Specification

To represent the model in source code, distinct object-oriented structures will be defined that map to the model semantics. Because they are part of arbitrary source code, arbitrary state spaces will exist beside the well-defined model information. Hence the model must also define interfaces between state machine structures and other source code to pass program execution control and variable values and thus hide the application logic.

So the "Embedded Model" is defined as $\Delta = \{Actor, \Sigma, \Theta, \Lambda, \Phi, \Psi\}$ with

- $Actor$ a facade type representing application logic which is invoked during transitions.
- $\Sigma = \{\sigma_s | s \in S\}$ a set of unique identified types that represent states.
- $\Theta = \{\theta_{t,\sigma} | t \in T, \sigma \in \Sigma\}$ a set of methods in state type σ, each representing a transition t.
- Λ an interface defining methods $\{\lambda_v | v \in V\}$ that return the current value for a variable v.
- $\Phi = \{\phi_t | t \in T\}$ a set of methods that implement guard checks for transitions.
- $\Psi = \{\psi_t | t \in T\}$ a set of methods that implement update checks for transitions.

State types implement an interface which defines no methods but allows to type-safely distinguish between state types and other types. Transition methods are designated with meta data that refers to the target state, guard and update implementations. They have no return type and take as parameter an *Actor* instance. The *Actor* type itself has arbitrary, application-specific content and is treated as a black box. Transition methods only make calls to methods provided by the *Actor* instance and therefore respect the conceptual separation between model and code.

The methods in Λ may query the application logic for any application variable at any point of time, but must never manipulate the application state. This way Θ and Λ are the only well-defined interfaces between model and application source code that allow manipulation and query of the application state.

Each ϕ_t returns true iff the pre-conditions of the guard hold. Due to their simple structure described above, guards can be mapped to logical and arithmetical expressions in the source code. Each variable v_n used in guards is represented as the according call of the method λ_n. Obviously the simplest possible guard is that there is none, in which case the source code method instantly returns true.

Each ψ_t returns true iff the variable updates interpreted as post-conditions hold. Parameters taken by this method are two instances of Λ to allow comparisons, one granting access to the current values and one containing cached variable values from the point in time before the transition fired. Since the method does not perform an actual update but validates the state, variable updates are represented similar to guards as logical and arithmetical expressions. Each single value update is replaced by a test for equality and each range update by a pair of comparisons with lower and upper bound. If an update should be left unchecked, the method can return true instantly.

3.2 Representation in JAVA

For an implementation of the concept sketched above, JAVA as a widespread object-oriented programming language and run time environment was chosen. The JAVA-specific constructs and conventions are shortly outlined here. The approach is not limited to JAVA since we can assume that similar concepts exist in other modern object-oriented languages too. A subset of the available declarative structures [9] is used, namely classes, interfaces, methods and annotations [10]. In combination with generic types the latter ensure type safety both for source code and meta information and thus facilitate an accurate source code creation by the developer.

State types are classes that implement the interface `IState`. All methods in the state classes are treated as transitions when decorated with the `@Transition` annotation. It contains an attribute `target` to denote the transition's target state class and an attribute `contract` that refers to a class containing guard and update methods. Λ is realized as an interface providing `get`-methods for each variable λ_v. The interface itself and its implementation are provided by the application developer. The contents of these methods are black boxes, too. It is up to the programmer to ensure that no manipulations of variable values happen when one

of these methods is called. Guards and updates for a transition are located in classes implementing the interface `IContract` with the generic type of Λ. It defines the according methods `checkCondition` and `validate` which return a boolean value and take parameters of the Λ type.

3.3 Model Execution

To execute the state machine model an execution component is required that walks through the state machine by interpreting state class declarations and transitions annotations. In each state the guard methods for each transition method are invoked to determine which transition will fire. Accordingly a transition method itself will be invoked. To start model execution the application passes three parameters to the execution component: The initial state class, the *Actor* instance and the Λ implementation. All other parts of the state machine structure are inferred from these and instantiated on demand.

To save resources, update checks are only enabled in a "debug" mode. In this case the current variable values are cached before a transition fires and afterwards provided to the update method together with the most recent variable values. For this purpose a fourth parameter is passed to the execution component, the Λ interface class, which is needed for dynamic instantiation of this type in Java for update methods. In summary, the execution component can access all information related to the state machine model at run time: States, transitions, variable values and their use in guards and updates. This way it is possible to monitor the state machine operation in real-time or to log the information and make activities traceable afterwards with only a few modifications.

3.4 Model Extraction for Design Time Analysis

For design time the Embedded Model is mapped to representations used in modelling tools. Because of the different emphases of existing modeling and verification tools, this cannot be done as universal as for object-oriented structures. Nevertheless the description of states, transitions and variables follows the general concepts of state machines and should hence be directly compatible with any modeling tool. On the other hand it has to be taken into account that the general theoretical concept of state machines is realized in different ways in common modeling techniques [11]. The example presented in section 2 showed the extracted model in the syntax of Uppaal, which is one sample output for a tool-specific mapping. When selecting an actual modeling tool and formulating the necessary mapping, it must be carefully examined whether the chosen tool provides a syntax powerful enough to express the semantics of guards and updates described above. Checking guards by evaluating variables, logical operators, arithmetic operators and comparators to boolean values can be assumed to be possible in most cases. Updating variables with single values, obtained from variables and arithmetic operations, is a standard technique, too. A range update is interpreted as a random choice of a value from the given interval followed by an update of the variable with this non-deterministic value. More precisely,

the question whether a tool supports range updates is the question whether it supports non-deterministic choices and allows to merge states defining the range of values for a variable into one single state. In our example, UPPAAL supports range updates for variables based on non-deterministic choices. Some minor challenges regarding naming were solved in this example, too. The data type `boolean` is named `bool` in UPPAAL and the `get`-prefix of all variable methods is stripped for better readability.

For this contribution, the model extraction was performed by graph transformations, based on the abstract syntax trees of JAVA and the DOM tree of the UPPAAL data format. Triple Graph Grammars [12] can be applied here for parallel transformations of source code and tool data format with the general state machine model as mapping schema. The detailed description of this graph grammar is beyond the scope of this paper.

4 Discussion

It is important to notice that our approach inverts the traditional direction of model-to-code generators. There is no model that is manipulated at design time and transformed into source code from time to time. Instead there is a permanent model representation in the source code, which is extracted for analysis within modelling tools from time to time. On the one hand this eliminates any effort to maintain and merge different abstraction layers. On the other hand, the chosen approach is not independent from programming languages and execution environments, in our case JAVA, as it is possible when using some other model-driven development technologies [1]. Hence our future work at the tool level aims to enable permanent partial transformations in real-time and hence parallel development of source code and external model representation. At the conceptual level we plan to realize more transformations from model to tools, e.g. into UML state chart diagrams [13] or the CADENCE SMV model checker [14].

The execution component benefits from the permanent representation of the model in code structures. Because of this the actual work done by the execution component is limited to class instantiations and method invocations. Since all dynamic functionality is contained in the invoked methods, the execution is very efficient as regular JAVA code is executed. At the same time the state machine model integrates in arbitrary business logic without enforcing restraints on the non-model code. On the other hand, the developer has to take care to organize the source code accordingly: The approach will only work if the clear separation is maintained and only valid expressions are used in methods whose content is interpreted, i.e. guard and update methods. To detect errors here is possible only if the model is interpreted at design time.

At a more general conceptual level, we aim to analyze the application of our general concept to domains and modeling methods other than state machines. Especially in these cases additional benefits can be expected for larger projects, because one change in source code may influence more than one embedded model.

5 Related Work

The attribute-oriented programming approach [15] with similar use of meta data in code structures has been explored to map UML models to code structures [16]. However, this does not leverage the principle of having only one representation for model and source code and does not avoid round trip engineering. The same applies to Framework Specific Modeling Languages [17], which could be of use if a state machine framework would control the application state. The concept of "executable UML" [7] tries to overcome inconsistencies between different representations by the use of automated transformations or by defining a primary representation that may generate and override all other representations. Our approach uses automated transformations to create the model from the source code and vice versa, but inside the source code the model is combined with non-model parts of the application, thus enabling a seamless integration into larger applications.

Different to the Java Modeling Language (JML) [18], which offers a huge syntax for specification annotations, we do not aim to present a notation for the specification of all possible system models. This applies also to the approach to use Smalltalk with it's introspection capabilities as a meta language [19]. Contrary to Java PathFinder [20] our approach does not consider a whole application as the model, but only selected parts of it. Hence our approach can be more complete and formally founded and thus be used for explicit representation and validation in this limited domain of state machine specifications.

6 Conclusion

In this contribution we proposed to embed state machine model semantics in source code structures and extract concrete model representations on demand. The model execution and extraction components have been outlined. As shown by example, we can extract a complete state machine representation from given Java source code. All of the source code structures in the Embedded Model are used without change to execute, monitor and debug the model at run time. Hence the objective to let application development in a larger context happen simultaneously to model specification, validation and simulation for parts of the application without double effort to maintain two abstraction levels is fulfilled. So we can state that our approach can effectively be used to avoid maintaining and merging different abstraction layers.

References

1. Brown, A.W., Iyengar, S., Johnston, S.: A Rational approach to model-driven development. IBM Systems Journal **45**(3) (2006) 463–480
2. Tichy, M., Giese, H.: Seamless UML Support for Service-based Software Architectures. In Guefi, N., Artesiano, E., Reggio, G., eds.: Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003,

Luxembourg. Volume 2952 of Lecture Notes in Computer Science., Springer-Verlag (November 2003) 128–138
3. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. (2005) 476–491
4. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development. (2004)
5. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal **45**(3) (2006) 451–461
6. Vokáč, M., Glattetre, J.M.: Using a Domain-Specific Language and Custom Tools to Model a Multi-tier Service-Oriented Application – Experiences and Challenges. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. (2005) 492–506
7. Mellor, S.J., Balcer, M.J.: Executable UML. Addison-Wesley (2002)
8. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer **1**(1–2) (Oct 1997) 134–152
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The 3rd Edition. Addison-Wesley Professional (2005)
10. Sun Microsystems, Inc.: JSR 175: A Metadata Facility for the Java™Programming Language http://jcp.org/en/jsr/detail?id=175.
11. Crane, M.L., Dingel, J.: UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of LNCS. (2005) 97–112
12. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science. Volume 903 of LNCS. (1994)
13. OMG: UML 2.0 superstructure specification. Technical report, Object Management Group (2004)
14. McMillan, K.: The Cadence SMV Model Checker http://www.kenmcmil.com/smv.html.
15. Schwarz, D.: Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. ONJava.com (June 2004)
16. Wada, H., Suzuki, J.: Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Briand, L.C., Williams, C., eds.: MoDELS. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 584–600
17. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. [21] 692–706
18. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In Kilov, H., Rumpe, B., Simmonds, I., eds.: Behavioral Specifications of Businesses and Systems, Kluwer (1999) 175–188
19. Ducasse, S., Gîrba, T.: Using Smalltalk as a Reflective Executable Meta-language. [21] 604–618
20. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal **10**(2) (2003)
21. Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006)

Model-Based Traces ⋆

(preliminary version)

Shahar Maoz

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
shahar.maoz@weizmann.ac.il

Abstract. We introduce *model-based traces*, which trace behavioral models of a system's design during its execution, allowing to combine model-driven engineering with dynamic analysis. Specifically, we take visual inter-object scenario-based and intra-object state-based models (sequence charts and statecharts) used for a system's design, and follow their activation and progress as they come to life at runtime, during the system's execution. Thus, a system's runtime is recorded and viewed through abstractions provided by behavioral models used for its design. We present two example applications related to the automatic generation and visual exploration of model-based traces and suggest a list of related challenges.

1 Introduction

Transferring model-driven engineering artifacts and methods from the early stages of requirements and specification, during a system's design, to the later stages of the lifecycle, where they would aid in the testing, analysis, maintenance, evolution, comprehension, and manipulation of running programs, is an important challenge in current model-driven engineering research.

In this paper, as a means towards this end, we introduce *model-based traces*, which trace behavioral models from a system's design during its execution, allowing to combine model-driven engineering with dynamic analysis. Specifically, we take visual inter-object scenario-based and intra-object state-based models (sequence diagrams and statecharts) used for a system's design, and follow their activation and progress as they come to life at runtime, during the execution of the system under investigation. Thus, a system's runtime is recorded and viewed through abstractions provided by models used for its design.

An important feature of model-based traces is that they provide enough information to reason about the executions of the system and to reconstruct and replay an execution (symbolically or concretely), exactly at the abstraction level defined by its models. This level of model-based reflection seems to be a necessary requisite for the kind of visibility into a system's runtime required for model-based dynamic analysis and adaptation.

Additional features worth noting. First, model-based traces can be generated and defined based on partial models; the level of abstraction is defined by the

⋆ This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

modeler. Second, the models used for tracing are not necessarily reflected explicitly in the running program's code; rather, they define a separate viewpoint, which in the process of model-based trace generation is put against the concrete runtime of the program under investigation. Third, the same concrete runtime trace may result in different model-based traces, based on the models used for tracing; and vice versa, different concrete runtime traces may result in equal model-based traces, if the concrete runs are equivalent from the more abstract point of view of the model used for tracing.

In the next section we briefly introduce, informally define, and discuss the format and features of model-based traces, using a simple example. We then present two example applications related to the automatic generation and visual exploration of model-based traces. Finally, we suggest a list of related challenges.

2 Model-Based Traces

The use of system's execution traces for different analysis purposes requires different levels of abstraction, e.g., recording CPU register assignments, recording virtual machine commands, or recording statements at the code level. We suggest a higher level of abstraction over execution traces, based on behavioral models typically used for a system's design, such as sequence diagrams and statecharts.

In this work we present two types of model-based traces, inter-object scenario-based traces and intra-object state-based traces. Additional types may be created by combining variants of the two or using other modeling techniques[1].

Given a program P and a behavioral model M, a model-based execution trace records a run r of P at the level of abstraction induced by M. A unification mechanism is defined, which statically and dynamically maps concrete elements of the run to elements in the model. The type of the model used, the artifacts and their semantics, define the types of entries that appear in the model-based trace. We demonstrate our ideas using two concrete examples of a scenario-based trace and a state-based trace, taken from a small example system.

Note that although there are code generation schemes for the *execution* of the models we use, we do not, in general and in the example given here, consider tracing programs whose code was automatically generated from models. On the contrary, we believe that one of the strengths of our approach is that it can be applied to systems in general, not only to ones where the implementation explicitly reflects certain high-level models.

Also note that the model-based traces we present are not mere projections of the concrete runtime information onto some limited domain. Rather, we use *stateful* abstractions, where trace entries depend on the history and context of the run and the model; the model-based trace not only filters out irrelevant information but also adds model specific information (e.g., information about entering and exiting 'states' that do not appear explicitly in the program).

A small example Consider an implementation of the classic PacMan game. PacMan consists of a maze, filled with dots, power-ups, fruit and four ghosts. A

[1] In principle, any representation of an execution trace may be considered a model-based trace, depending on the definition of what constitutes a model.

human player controls PacMan, who needs to collect as many points as possible by eating the objects in the maze. When a ghost collides with PacMan, it loses a life. When no lives are left, the game is over. However, if PacMan eats a power-up, it is temporarily able to eat the ghosts, thus reversing roles. When a ghost is eaten, it must go back to the jail at the center of the maze before leaving again to chase PacMan. When all dots are eaten, the game advances to the next – more difficult – level. We consider the PacMan game to be a well-known, intuitive, relatively small and yet complex enough reactive system, hence a good choice for the purpose of demonstrating model-based traces in this paper.

2.1 Scenario-based models

For inter-object scenario-based modeling, we use a UML2-compliant variant of Damm and Harel's *live sequence charts* (LSC) [4,9]. Roughly, LSC extends the partial order semantics of sequence diagrams in general with a universal interpretation and must/may (hot/cold) modalities, and thus allows to specify scenario-based liveness and safety properties. Must (hot) events and conditions are colored in red and use solid lines; may (cold) events and conditions are colored in blue and use dashed lines. A specification typically consists of many charts, possibly interdependent, divided between several use cases (our small PacMan example has 9 scenarios divided between 3 use cases).

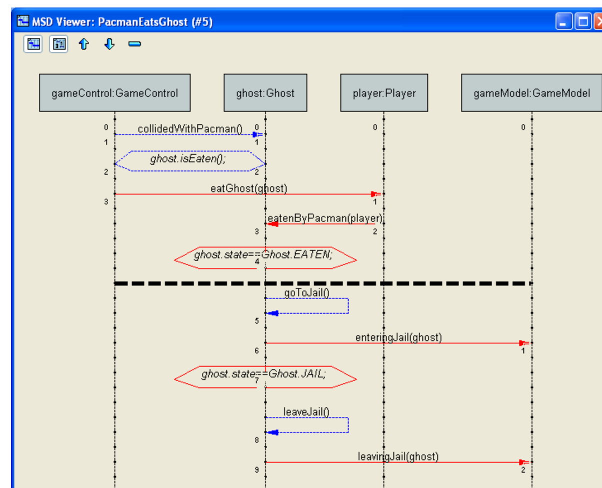

Fig. 1. The LSC for `PacManEatsGhost` with a cut displayed at (3,4,2,0).

Fig. 1 shows one LSC taken from our example model of PacMan. Vertical lines represent specific system objects and time goes from top to bottom. Roughly, this scenario specifies that *"whenever a `gameControl` calls a ghost's `collidedWithPacman()` method and the ghost's `isEaten()` method evaluates to TRUE, the `gameControl` must tell the `player` (PacMan) to eat the `ghost`, the `player` must tell the `ghost` it has been eaten, and the ghost's `state` must change to `EATEN`. Then, if and when the `ghost` goes to jail it must tell the `gameModel` it has gone there and its `state` should change to `JAIL`, etc..."* Note the use of hot

'must' elements and cold 'may' elements. Also, note the use of symbolic instances (see [15]): the lifeline representing `ghost` may bind at runtime to any of the four ghosts (all four are instances of the class `Ghost`).

An important concept in LSC semantics is the *cut*, which is a mapping from each lifeline to one of its locations (note the tiny location numbers along the lifelines in Fig. 1, representing the state of an active scenario during execution). The cut `(3,4,2,0)`, for example, comes immediately after the hot evaluation of the ghost's state. A cut induces a set of enabled events — those immediately after it in the partial order defined by the diagram. A cut is hot if any of its enabled events is hot (and is cold otherwise). When a chart's minimal event occurs, a new instance of it is activated. An occurrence of an enabled method or `true` evaluation of an enabled condition causes the cut to progress; an occurrence of a non-enabled method from the chart or a `false` evaluation of an enabled condition when the cut is *cold* is a *completion* and causes the chart's instance to close gracefully; an occurrence of a non-enabled method from the chart or a `false` evaluation of an enabled condition when the cut is *hot* is a *violation* and should never happen if the implementation is faithful to the specification model. A chart does not restrict events not explicitly mentioned in it to occur or not to occur during a run (including in between events mentioned in the chart).

2.2 Scenario-based traces

Given a scenario-based specification consisting of a number of LSCs, a *scenario-based trace* includes the activation and progress information of the scenarios, relative to a given program run. A trace may be viewed as a projection of the full execution data onto the set of methods in the specification, plus, significantly, the activation, binding, and cut-state progress information of all the instances of the charts (including concurrently active multiple copies of the same chart). Thus, our scenario-based traces may include the following types of entries:

- **Event occurrence** representing the occurrence of an event. Events are timestamped and are numbered in order of occurrence. Only the events that explicitly appear in one of the scenarios in the model are recorded in the trace (one may add identifiers of participating objects, i.e., caller and callee, and parameter values). The format for an event occurrence entry is:
  `E: <timestamp> <event no.>: <event signature>`
- **Binding** representing the binding of a lifeline in one of the active scenario instances to an object. Its format is:
  `B: <scenario name>[instance no.] lifeline <no.> <- <object identifier>`
- **Cut change** representing a cut change in one of the active scenario instances. Its format is:
  `C: <scenario name>[instance no.] <cut tuple> [Hot|Cold]`
- **Finalization** representing a successful completion or a violation in an active scenario instance. Its format is:
  `F: <scenario name>[instance no.] [Completion|Violation]`

Fig. 2 shows an example short snippet from a scenario-based trace of PacMan. Note the different types of entries that appear in the trace.

```
...
E: 1172664920526 64: void pacman.classes.Ghost.slowDown()
B: PowerUpEaten[1] lifeline 6 <- pacman.classes.Ghost@7e987e98
B: GhostStopsFleeing[7] lifeline 1 <- pacman.classes.Ghost@7e987e98
C: GhostStopsFleeing[7] (0,1) Hot
C: GhostFleeing[7] (1,3) Hot
E: 1172664920526 65: void pacman.classes.GameControl.ghostSlowedDown(Ghost) pacman.classes.Ghost@7e987e98
B: GhostStopsFleeing[7] lifeline 0 <- pacman.classes.GameControl[panel0,0,0,600x600,layout=...
C: GhostStopsFleeing[7] (1,2) Cold
C: GhostFleeing[7] (2,4) Cold
E: 1172664920526 66: void pacman.classes.GameModel.resetGhostPoints()
C: PowerUpEaten[1] (1,2,6,1,1,1,1) Cold
F: PowerUpEaten[1] Completion
E: 1172664921387 67: void pacman.classes.Fruit.enterScreen()
B: PacmanEatsFruit[0] lifeline 2 <- pacman.classes.Fruit@3360336
C: PacmanEatsFruit[0] (0,0,1,0) Hot
C: PacmanEatsFruit[0] (0,0,2,0) Cold
E: 1172664923360 68: void pacman.classes.Ghost.collidedWithPacman()
B: PacmanEatsGhost[2] lifeline 1 <- pacman.classes.Ghost@7d947d94
B: PacmanEatsGhost[2] lifeline 0 <- pacman.classes.GameControl[panel0,0,0,600x600,layout=...
C: PacmanEatsGhost[2] (1,1,0,0) Hot
C: PacmanEatsGhost[2] (1,2,0,0) Hot
C: GhostEatsPacman[2] (0,1,1,0) Cold
F: GhostEatsPacman[2] Violation
...
```

Fig. 2. Part of a textual representation of a scenario-based trace of PacMan.

2.3 State-based models

For intra-object state-based modeling, we use UML state machines (that is, the object based variant of Harel statecharts [7]). For lack of space, we assume the reader is partly familiar with the syntax and semantics of statecharts in general, at least to the level that allows to understand our example.

Fig. 3 shows an example statechart taken from a model of PacMan. It shows part of a statechart for the class `Ghost`.

2.4 State-based traces

Given a state-based specification consisting of a number statecharts, a *state-based trace* includes the creation and progress information of the statecharts, relative to a given program run. The trace includes information on events, guards evaluation, and the entering and exiting of states in all instances of the statecharts (including concurrently running instances of the same statechart). Thus, our state-based traces may include the following types of entries:

– **State entered** representing a statechart entering a state. The format is:

  `EN: <class_name>[instance no.] Entered state <state full name>`

– **State exited** representing a statechart existing a state. The format is:

  `EX: <class_name>[instance no.] Exited state <state full name>`

– **Event occurrence** representing the occurrence of an event. Events are timestamped and are numbered in order of occurrence. Only the events that explicitly appear in one of the statecharts in the model are recorded in the trace. One may optionally add guards evaluation. The format is:

  `EV: <timestamp> <event no.>: <event signature>`

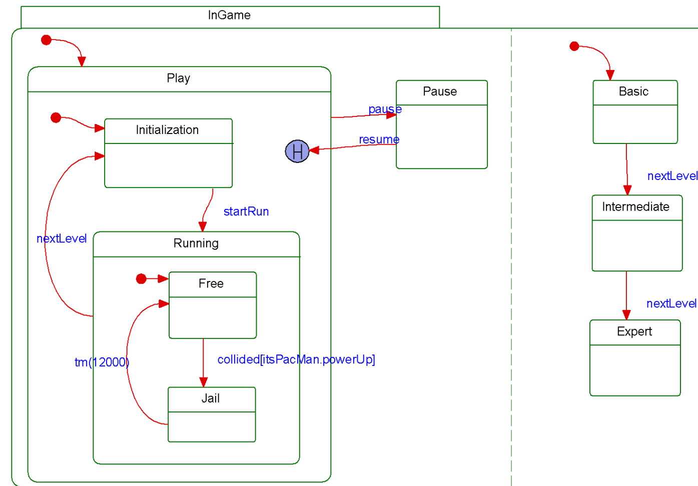

Fig. 3. Part of the `Ghost` statechart in the PacMan model.

Fig. 4 shows a snippet from a state-based trace of PacMan involving a number of statecharts. Note the different types of entries that appear in the trace.

We remark that the above scenario-based and state-based trace formats are presented as examples. Depending on the application, the trace generation mechanism available, and the kind of analysis and reasoning intended for the model-based traces, one may consider different formats, different entry types, different levels of succinctness etc. For example, whether to document the values of guards or the concrete values of parameters depends on the specific application and expected usage of the model-based trace.

3 Example Applications

We give a short overview of two example applications related to the generation of model-based traces and to their visualization and exploration.

3.1 Generating model-based traces

S2A [8] (for Scenarios to Aspects) is a compiler that translates live sequence charts, given in their UML2-compliant variant using the *modal* profile [9], into AspectJ code [1], and thus provides full code generation of reactive behavior from visual declarative scenario-based specifications. S2A implements a compilation scheme presented in [13]. Roughly, each sequence diagram is translated into a *scenario aspect*, implemented in AspectJ, which simulates an automaton whose states correspond to the scenario cuts; transitions are triggered by AspectJ pointcuts, and corresponding advice is responsible for advancing the automaton to the next cut state.

```
...
EV: 45632290 874: Ghost[3].collided
EX: Ghost[3] Exited state Ghost.InGame.InPlay.Play.Running.Free
EN: Ghost[3] Entered state Ghost.InGame.InPlay.Play.Running.Jail
EV: 45644272 875: Ghost[2].collided
EX: Ghost[2] Exited state Ghost.InGame.InPlay.Play.Running.Free
EN: Ghost[2] Entered state Ghost.InGame.InPlay.Play.Running.Jail
EV: 45644290 876: Ghost[3].timer
EX: Ghost[3] Exited state Ghost.InGame.InPlay.Play.Running.Jail
EN: Ghost[3] Entered state Ghost.InGame.InPlay.Play.Running.Free
EV: PacMan[1] 877: Pacman[1].complete
EX: PacMan[1] Exited state PacMan.InPlay.Play
EN: PacMan[1] Entered state PacMan.InPlay.LevelInitalization
EV: 45664403 878: Ghost[1].nextLevel
EX: Ghost[1] Exited state Ghost.InGame.InPlay.Play.Running.Free
EX: Ghost[1] Exited state Ghost.InGame.Levels.Basic
EN: Ghost[1] Entered state Ghost.InGame.InPlay.Play.Initalization
EN: Ghost[1] Entered state Ghost.InGame.Levels.Intermediate
EV: 45664405 879: Ghost[2].nextLevel
EX: Ghost[2] Exited state Ghost.InGame.InPlay.Play.Running.Jail
EX: Ghost[2] Exited state Ghost.InGame.Levels.Basic
EN: Ghost[2] Entered state Ghost.InGame.InPlay.Play.Initalization
EN: Ghost[2] Entered state Ghost.InGame.Levels.Intermediate
EV: 45664408 880: Ghost[3].nextLevel
...
```

Fig. 4. Part of a textual representation of a state-based trace of PacMan.

Most important in the context of this paper, though, is that in addition to scenario-based execution (following the play-out algorithm of [10]), S2A provides a mechanism for scenario-based monitoring and runtime verification. Indeed, the example scenario-based trace shown in Fig. 2 is taken from an actual execution log of a real Java program of the PacMan game adapted from [3], (reverse) modeled using a set of live sequence charts (drawn inside IBM Rational SA [2] as modal sequence diagrams), and automatically instrumented by the AspectJ code generated by S2A. More on S2A and its use for model-based trace generation can be found in `http://www.wisdom.weizmann.ac.il/~maozs/s2a/`.

3.2 Exploring model-based traces

The Tracer [14] is a prototype tool for the visualization and interactive exploration of model-based traces. The input for the Tracer is a scenario-based model of a system given as a set of UML2-compliant live sequence charts, and a scenario-based trace, generated from an execution of the system under investigation.

Fig. 5 shows a screenshot of the main view of the Tracer, displaying a scenario-based model and trace similar to the one shown in Fig. 2. Roughly, the main view is based on an extended hierarchical Gantt chart, where time goes from left to right and a two-level hierarchy is defined by the containment relation of use cases and sequence diagrams in the model. Each leaf in the hierarchy represents a sequence diagram, the horizontal rows represent specific active instances of a diagram, and the blue and red bars show the duration of being in a specific cold and hot relevant cuts. The horizontal axis of the view allows to follow the progress of specific scenario instances over time, identify events that caused progress, and locate completions and violations. The vertical axis allows

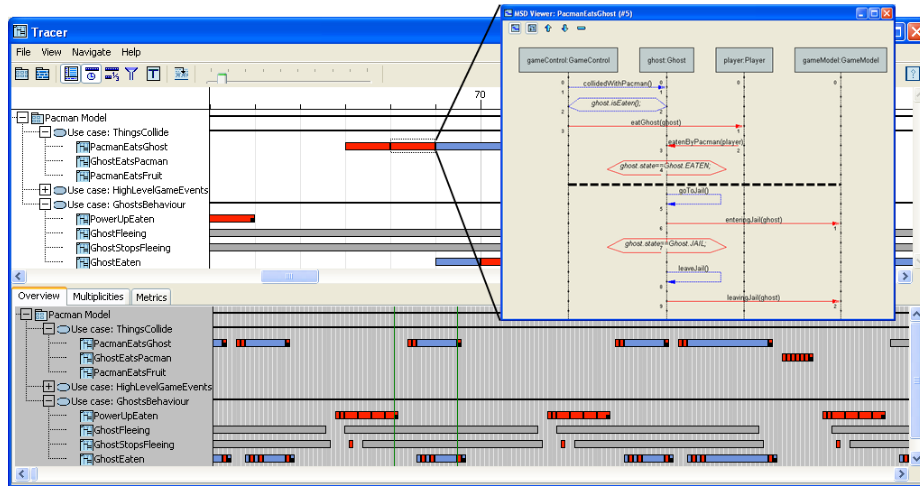

Fig. 5. The Tracer's main view, an opened scenario instance with its cut displayed at (3,4,2,0), and the Overview pane (at the bottom). The example trace and model are taken from an implementation of the PacMan game, see [14].

a clear view of the synchronic characteristic of the trace, showing exactly what goes on, at the models abstraction level, at any given point in time.

When double-clicking a bar, a window opens, displaying the corresponding scenario instance with its dynamic cut shown in a dashed black line. Identifiers of bound objects and values of parameters and conditions are displayed in tooltips over the relevant elements in the diagram. In addition, one can travel back and forth along the cuts of the specific instance (using the keyboard or the arrows in the upper-left part of the window). Multiple windows displaying the dynamic view of different scenario instances can be opened simultaneously to allow for a more global synchronic (vertical) view of a specific point in the execution, or for a diachronic (horizontal) comparison between the executions of different instances of the same scenario at different points in time during the execution.

Note the Overview pane (bottom of Fig. 5), which displays the main execution trace in a smaller pixel per event scale, and the moving window frame showing the borders of the interval currently visible in the main view. The Overview allows to identify high level recurring behavioral patterns, at the abstract level of the scenarios in the model. Additional views are available, supporting multiplicities, event-based and real-time based tracing, and the presentation of various synchronous metrics (e.g., how many scenarios have been affected by the most recent event?). Overall, the technique links the static and dynamic aspects of the system, and supports synchronic and diachronic trace exploration. It uses overviews, filters, details-on-demand mechanisms, multi-scaling grids, and gradient coloring methods.

The Tracer was first presented in [14]. More on the Tracer, including additional screenshots and screencasts can be found in `http://www.wisdom.weizmann.ac.il/~maozs/tracer/`.

4 Related work

We briefly discuss related work. Generating model-based traces requires an observer with monitoring and decision-making capabilities; a so called 'runtime awareness' component (see, e.g., [5,11]). However, while model-based traces can be used for error detection and runtime verification, the rich information embedded in them supports more general program comprehension and analysis tasks and allows the reconstruction and symbolic replay of a program's run at the abstraction level defined by the model used for tracing.

The use of AOP in general and AspectJ in particular to monitor program behavior based on behavioral properties specified in (variants of) LTL has been suggested before (see, e.g., [5,16]). As LSCs can be translated into LTL (see [12]), these work have similarities with our use here of S2A. Like [16], S2A automatically generates the AspectJ code which simulates the scenario automaton (see [13]). Unlike both work however, S2A outputs a rich trace reflecting state changes and related data (binding etc.), to serve our goal of generating model-based traces that allow visibility and replaying, not only error detection.

Many work suggest various trace generation and visual exploration techniques (e.g., for a survey, see, [6]). Most consider code level concrete traces. Some attempt to extract models from these traces. In contrast, model-based traces use an abstraction given by user-defined models. They are generated by symbolically running these models simultaneously with a concrete program execution.

5 Discussion and Challenges for Future Work

We introduced model-based traces and presented two example applications. The focus of model-based traces is on providing visibility into an execution of a program at the abstraction level defined by a model, enabling a combination of dynamic analysis and model-driven engineering. Below we discuss our approach and list related challenges.

Trace generation S2A provides an example of a model-based trace generation technology, based on programmatically generated aspects. Two major advantages of this approach are that the monitoring code is automatically generated from the models, and that the code of the system under investigation itself is oblivious to the models 'watching' it. Related challenges include minimizing runtime overhead, scalability in terms of trace length and model size, and the application of similar technology to domains where aspect technology is not readily available (e.g., various embedded or distributed systems).

Analysis and reasoning We consider the development of analysis methods for model-based traces. For example, define and measure various vertical and horizontal metrics (e.g., 'bandwidth', state / transition coverage per trace per model, how many times was each state visited), abstraction and refinement operators (e.g., hide events and keep states, hide sub states of composite states), ways to represent and compare different model-based runtime configurations ('snapshots', perhaps most important for dynamic adaptation), or ways to align and compare between traces of different runs of the same system, or very similar

runs of different versions of the same system. In addition, we consider additional types of abstractions over the same models, e.g., real-time based vs. event-based trace representation (as is supported by the Tracer (see [14])). Also, an agreeable, common representation format for model-based traces, where applicable (e.g., for specific types of models), should perhaps be defined and agreed upon, so that not only models but also their traces may be exchanged between tools in a standard format like XMI.

Visualization and interaction The visualization and interaction supported by the Tracer allows a human user to explore and analyze long and complex model-based traces that are otherwise very hard to handle manually in their textual form. Still, a lot more can be done on this front, from finding "economic" visualizations for model-based snapshots to animation to visual filters etc.

Acknowledgements I would like to thank David Harel, David Lo, Itai Segall, Yaki Setty, and the anonymous reviewers for comments on a draft of this paper.

References

1. AspectJ. http://www.eclipse.org/aspectj/.
2. IBM Rational Software Architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.
3. PacMan. Java implementation of the classic PacMan game. http://www.bennychow.com.
4. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
5. H. Goldsby, B. H. C. Cheng, and J. Zhang. AMOEBA-RT: Run-Time Verification of Adaptive Software. In *Models@run.time, MoDELS Workshops*, 2007.
6. A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON*, 2004.
7. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
8. D. Harel, A. Kleinbort, and S. Maoz. S2A: A Compiler for Multi-Modal UML Sequence Diagrams. In *FASE*, 2007.
9. D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.
10. D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and Systems Modeling (SoSyM)*, 2(2):82–107, 2003.
11. J. Hooman and T. Hendriks. Model-Based Run-Time Error Detection. In *Models@run.time, MoDELS Workshops*, 2007.
12. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *TACAS*, 2005.
13. S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.
14. S. Maoz, A. Kleinbort, and D. Harel. Towards Trace Visualization and Exploration for Reactive Systems. In *IEEE VL/HCC*, 2007.
15. R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *OOPSLA*, 2002.
16. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. *Electr. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.

Mutual Dynamic Adaptation of Models and Service Enactment in ALIVE*

Athanasios Staikopoulos[1], Sébastien Saudrais[1], Siobhán Clarke[1],
Julian Padget[2], Owen Cliffe[2] and Marina De Vos[2]

[1] Trinity College Dublin, Computer Science, Ireland
{Athanasios.Staikopoulos, Sebastien.Saudrais, Siobhan.Clarke}@cs.tcd.ie
[2] University of Bath, Computer Science, UK
{jap, occ, mdv}@cs.bath.ac.uk

Abstract. In complex service-oriented systems, a number of layers of abstraction may be considered, in particular the models of the organisations involved, how interactions are coordinated and the services which are used and made available, are all relevant to the construction of complex service-oriented systems. As each of these layers is built upon another there is a clear need to provide a maintenance mechanism, capable of maintaining consistency across the concepts used in each layer. In addition, over time designs may change because of the introduction of new requirements and the availability and capabilities of services may change due to implementation modifications or service failures, leading to the need to consider a two-way adaptation, namely between the system design and its run-time. The contribution of this paper is the description of our (novel) mutual adaptation mechanism and, using an industry scenario based on the proposed ALIVE framework, its illustration in use of the kinds of adaptation.

Keywords: Model-driven architecture, web services, workflows, monitoring, adaptation.

1 Introduction

Today's software systems are becoming increasingly large and complicated. They are built upon many different technologies where a variety of abstraction layers are utilized, making it difficult for software engineering methodologies to support properly the various stages of their life-cycle, including design, implementation of artefacts and actual execution. Consequently, there is a clear need to develop maintenance and monitoring mechanisms allowing the dynamic adaptation, reconfiguration and self-management of such systems. It becomes increasingly clear that such mechanisms can provide a fundamental framework, where other more elaborate mechanisms can be established moving systems towards the vision of

* This work has been carried out in the framework of the FP7 project ALIVE IST-215890, which is funded by the European Community. The author(s) would like to acknowledge the contributions of his (their) colleagues from ALIVE Consortium (http://www.ist-alive.eu)

autonomic computing [1], where under certain circumstances a system may (re-) configure itself and adapt automatically to changing environments.

The work described in this paper is carried out in the context of the EU-funded ALIVE project [2, 3]. The premise behind the project is that current service-oriented architectures (SOAs) are typically incremental developments of existing Web service frameworks, making them fragile and inappropriate for long-term deployment in changing environments. Our proposed solution is to utilize the rich body of experience found in human organisations through the formalization of organisational theory and the coordination mechanisms that underpin the interactions between the entities. This provides us with a range of strategies that have been tried-and-tested in (human) social and economic contexts and that, with the provision of sufficient appropriate information about the state of the environment and the enactment of a workflow, can be applied to the dynamic adaptation of SOAs. A key element of our solution is the use of model-driven architectural descriptions of the SOA design – representing the organisational and coordination artefacts mentioned earlier – that admit formal adaptation and are thus able to capture and reflect changes in the deployed system.

In this paper we propose a bidirectional adaptation approach for maintaining design models with their run-time execution. The models visualising the service organisations and coordination as specified in ALIVE are used in a model-driven approach, while service enactment is a result of a model transformation process.

In SOA functional components are exposed as services, each of which is associated with an externalised description of the service's interface and functionality. These services are composed and linked in a loosely-coupled pattern in such a way that individual services may be replaced and re-used without modification. Current approaches to SOAs build on existing Web service (WS) technologies, such as SOAP, WSDL and BPEL to describe and execute service interactions. Given a set of services, process descriptions in the form of workflows may be constructed and executed using existing workflow interpreters, which take a given language such as BPEL and invoke services in accordance with the specified flow of control.

Model Driven Engineering (MDE) refers to the systematic use of models as primary artefacts for the specification and implementation of software systems. The Model Driven Development (MDD) methodology is based on the automatic creation of implementation artefacts from abstracted models via a predefined model transformation process. So far, model-driven approaches are primarily focused on the design, implementation and deployment stages of software development. However, MDD can similarly support the maintenance, requirements and testing phases. In those cases, MDD can be applied in the opposite direction, for the purpose of building or recovering high-level models from existing implementation artefacts to support round-trip engineering. Thus, it is possible to bridge the gap and provide consistency among design models and actual executions.

The remainder of the paper is organised as follows: Section 2 provides an overview of the research context. Section 3 presents our mutual adaptation approach for models and enactments. Section 4 highlights our approach with an example drawn from an ALIVE use-case scenario. Section 5, provides various discussion points and compares our approach with related work. Finally, section 6 outlines our conclusions and summarises the fundamental characteristics of our approach.

2 Dynamic Model Adaptation

Dynamic model adaptation refers to applying automated modifications on models often representing executing systems at run-time. Model-driven development often produces design artefacts that are lost during the execution and yet may be needed if the architect wants to change the actual execution when something goes wrong. The use of run-time models permits the complete or partial reuse of the current design models and their adaptation to the actual execution of systems. In particular [4] gives examples where run-time models can be useful in adaptation of systems. These examples are relevant to our two-way dynamic model adaptation mechanism.

The first case where run-time models are useful is the observation of the execution. The execution utilizes real code to perform the functions prescribed by the models. The use of a run-time model, based on the observation of the execution, allows for the creation of an abstract view of the execution, which in turn may be used by an adaptation module. The set of events which are observed in this process have to be generated from the design models.

The second case is the automatic adaptation of the system depending on the execution's observation. Patterns of adaption are usually defined by the architect during the design phase by taking account of some critical execution events. When a predefined set of events is triggered, the adaptation is performed on the run-time model and then changes are applied in the generated execution.

Finally, the third case is redesigning the actual execution using the run-time models. The architect, by looking at the run-time models, may decide to modify or add new functionalities to the system. These modifications are then transferred to the execution by production of run-time changes.

3 An Approach for Mutual Dynamic Adaptation

In this paper, we propose an approach for the dynamic adaptation of models and executables based on model transformations and the monitoring of the service enactment. The adaptation of models and executables is performed dynamically; both automatically and at run-time. Moreover, their dynamic adaptation is not based on the direct execution of models, so they are not compiled by model compilers and they do not run on specialised virtual machines - where executable models are monitored, but rather the adaptation is based on monitoring the enactment of native code that is the product of a model driven transformation process. Next, a monitoring mechanism monitors changes on service enactment and on design models by listening to specific significant events. Depending on the events generated the corresponding handling module is triggered to maintain/adapt the design models and generate the new enactment that will be loaded and executed from tools. The connectivity of external tools and the monitoring mechanism is maintained by the instrumentation framework. The approach is mutual, meaning that adaptations can be performed both a) from run-time execution to design models and b) from design models to run-time execution.

Another important characteristic that distinguishes our approach from others is that in our case model adaptations are applied both on structural models defining the

organisation of Multi Agent Systems (MAS) [5] and behavioural models defining their coordination. Furthermore, adaptations are applied on agent/service allocation and deployment, which are subject to various criteria such as availability of resources and generation of unexpected faults.

More specifically, our approach is influenced by the three levels identified in the ALIVE project, namely; the organisation, coordination and services. Each of the levels plays an important role in MAS. For example, organisation provides the structure, relation and rules of agents, coordination specifies the allowable patterns of interaction and services provide the rules of engagement in terms of services. This multi-layer conceptual separation of concerns provides a number of architectural advances, based on the fundamental concepts of decoupling and modularisation.

In order to reflect this architectural alignment within the ALIVE project the adaptation process has to cross both directions (bottom-to-top and top-to-bottom) in the multi level hierarchy. Thus, changes in the service level may require adaptations of the coordination model and in turn changes in the coordination model may require changes of the organisation structure. Very similarly, this adaptation dependency is implied in the opposite direction from organisation to coordination and services. In that way, the ALIVE architecture remains highly adaptive across its inner and cross levels. At implementation level, the dependency of inner adaptations is maintained by linking the Organisation, Coordination and Service handlers, whereas cross dependency via transformations.

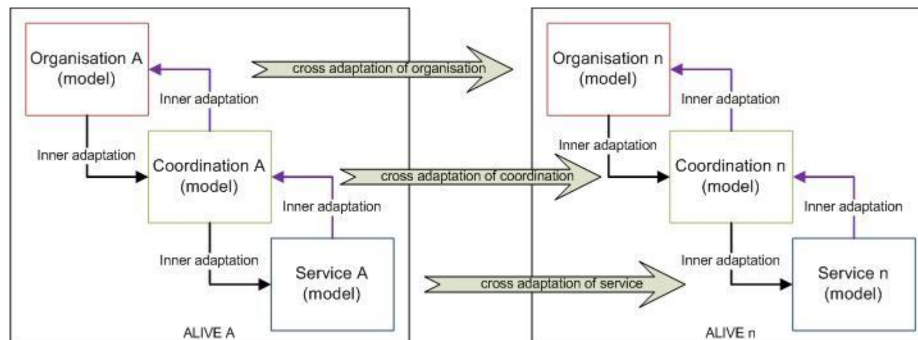

Fig. 1. Maintaining Multi-Levels of Model Adaptation

3.1 Adaptation steps and process

Conceptually, within MDE each of the ALIVE levels is formalised and represented with a corresponding metamodel. The models which are diagramming instances of the ALIVE metamodel are created by designers using specialised graphical tools. After models have been constructed, model transformations are defined to create executable process specifications in languages such as BPEL. Specialised tools (engines) can then load the executables and initiate the enactment of the modelled ALIVE scenario.

Process executions are instrumented with a monitoring framework, which listens for significant events during the execution of a given process. When a significant

event occurs the monitor is notified and the control is transferred on the corresponding handler. The handlers are interlinked to reflect the architectural dependencies among levels, and maintain the process of inner adaptation. Connectivity among external tools (engines) and the monitoring mechanism is maintained by a middleware instrumentation framework.

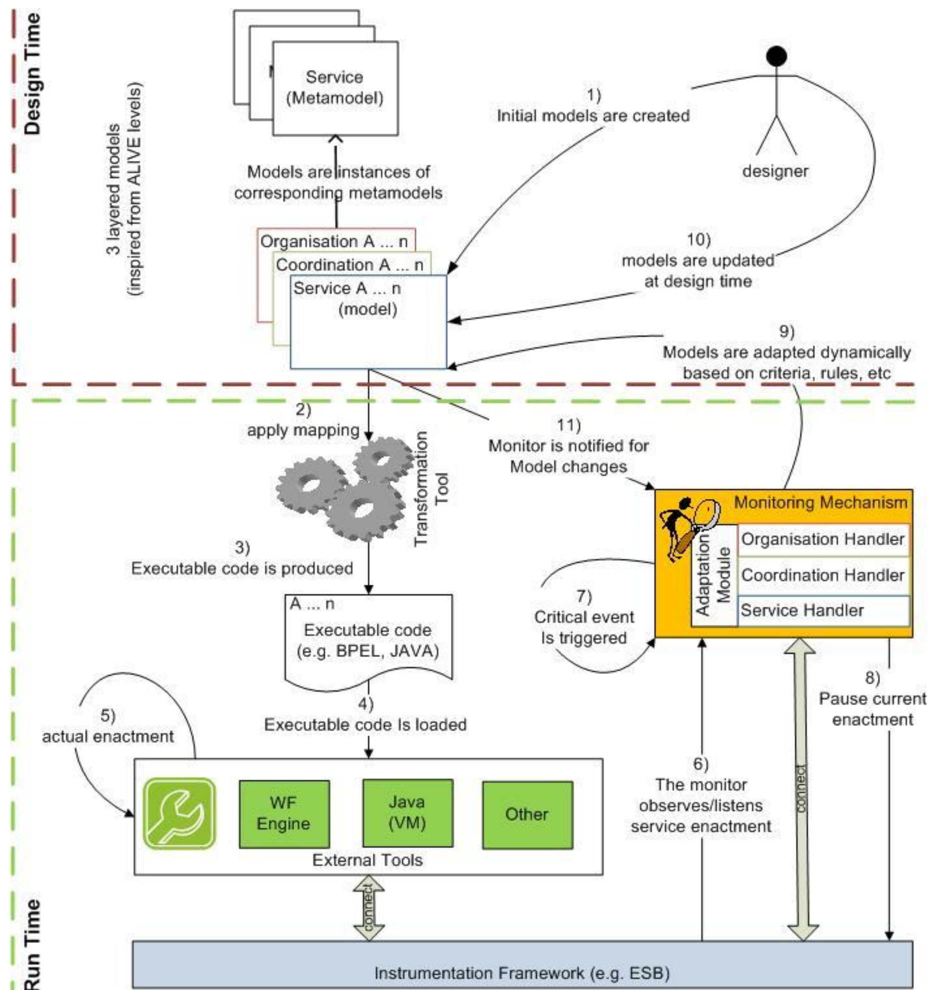

Fig. 2. Our Mutual, Multi-Layered Adaptation Approach.

The process steps can be distinguished into three phases as follows:

Initialisation phase: The initialisation phase corresponds to the design time and the generation of the executable code. The first step is the creation of the organisation, coordination and service models by the architect (1) using design tools. The models which are instances of the ALIVE metamodel depict a particular use case scenario such as Thales. At design-time the designer can also specify automatic execution

adaptations that will be executed by the adaptation module. The models are next sent to predefined model transformations (2) to automatically create executable code (3), such as BPEL and WSDL. Then, execution tools load the executable code and initiate enactment (4). During the execution (5), a monitor mechanism observes execution and listens for specific significant events (6) controlled by conditions, rules etc.

Model adaptations due to events/failures in service enactment: During the execution of the application, adaptations may occur depending on the significant events. Initial plans may not be possible to be performed due to limited availability of resources, failures and other external reasons. These (critical) events are captured by the monitoring mechanism and passed on the corresponding (organisation, coordination, service) model handler for an adaptation action (7) whereas the current service enactment is suspended (8). As a result, the corresponding model handler dynamically updates/adapts existing models to new ones (9). Depending on the rules, adaptations may be propagated internally between the successive inner levels of ALIVE. Once the new models are produced, the generation process produces new executions by using steps (2-3-4) and the service enactment restarts (5).

Adaptation of service enactment due to design alterations: Alternatively, adaptations can occur as a result of a manual modification of the models by the architect while service enactment (10). The monitor mechanism is notified for the model changes (11) and the current enactment is suspended (8). Once more new executable code is generated by steps (2-3-4) and an updated enactment restarts (5).

4 Applying the Approach with an ALIVE Scenario

At this point, we present how the two-way dynamic adaptation of models and service enactment is maintained with a motivation example. The example describes a crisis management scenario from THALES [6, 7] used in the context of ALIVE project [2]. More specifically, the scenario describes how the Dutch Ministry of Internal Affairs manages an emergency depending on the severity of an incident, by defining five GRIP levels of emergency handling. Each level specifies the tasks, roles, authorities and responsibilities of the members involved in the handling of an incident. For purposes of simplicity, in this paper we consider an emergency scenario scaled from GRIP 0 to 1. GRIP 0 describes how to handle a routine accident where no major coordination is required, whereas GRID 1 describes how many different authorities coordinate at an operational level.

4.1 Initialisation phase

Initially, at design time the organisation, coordination and service concepts of the THALES scenario are modelled at GRIP 0 level by the designer. In this example a combination of UML 2.0 diagrams are used to depict effectively these concepts with Class/Collaboration, Interaction and Component models respectively.

Organisation: At GRIP-0, the organisation consists of few structures. Most importantly, the *CrisisManagement* class has a *GripLevel* attribute to maintain the current state of the incident. *CrisisManagement* is related to at most one (see optional

cardinality [0..1]) *Ambulance*, *Fire_Fighting_Team* and *PoliceOfficer* classes. The *Handle_Incident* collaboration depicts how a *PoliceOfficer* playing the role of *securePlace*, an *Ambulance* by *provideTreatment* and a *Fire_Fighting_Team by extinguishFire* collaborate with one another to handle an incident.

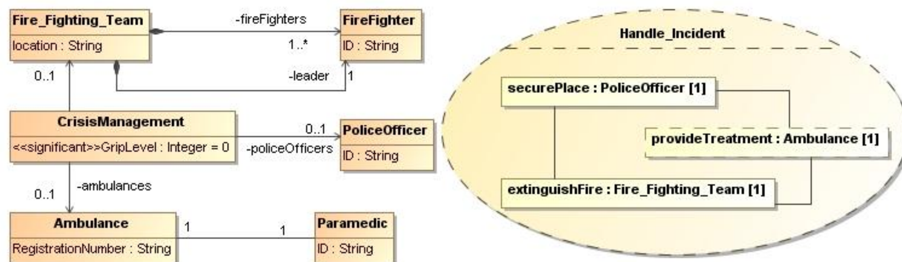

Fig. 3. Organisation models at GRIP 0

Coordination: At GRIP-0, the coordination (describes the possible interactions among members) for handling an incident is specified in a network-like relation. All parties have equal responsibility in resolving the situation and communicate via *inform* methods and exchange *incident* information.

Service: At GRIP-0, the services/agents are limited to those of a *FireService*, *PoliceService* and *AmbulanceService*. The services have to implement the interaction structures specified at coordination level and expose the relevant operations and interfaces.

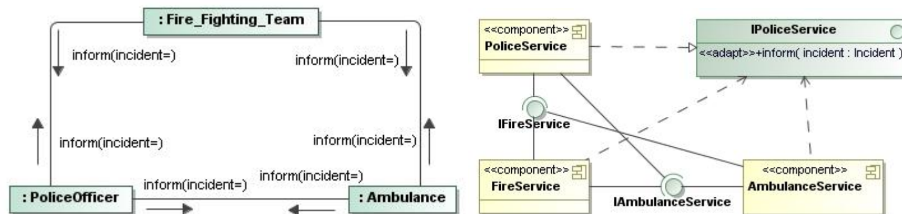

Fig. 4. Coordination (left) and Service (right) models at GRIP 0

Later the coordination patterns and interfaces will be transformed to corresponding Web service implementations for BPEL and WSDL via predefined model transformations. At this point we do not present the details of the transformation process, however there are many approaches in this regard see [8, 9]. Next the generated artefacts are loaded for execution to an execution engine such as Apache's Orchestration Director Engine (ODE) [10].

The significant events need to be marked with stereotypes and tag values on design models, so appropriate handlers can be created. For example, in fig.3 we have marked the property *GripLevel* of *CrisisManagement* as significant, so an appropriate handler can be created to monitor the state changes during enactment. Similarly, exceptions on interface operations can be marked as adapted, indicating that a handler needs to be generated and the path of enactment needs to be changed.

Finally, specific adaptation rules are defined by the designer and attached to models. These rules define the adaptation patterns to be followed in case of a significant event. The rules may be specified in a QVT-like language or refer to other implementations of ontological or rule-based languages. The handlers are capable to interpret these rules and perform the adaptations.

4.2 Model adaptations due to events/failures in service enactment

During the execution of the workflow, significant events may be triggered and processed by the monitoring mechanism. The events may propagate a series of inner adaptations from their corresponding handlers to design models as seen in chapter 3.

Thus, during the execution of the *PoliceService* by an agent, an error may occur due to some unavailable resources. In this case the models have to be adapted at run-time with new enactment plans which first need to be constructed. The adaptation process is directed by the adaptation pattern associated with the significant event and retrieved from the model. The pattern may be specified in model-driven native specification (QVT based) or other (rule-based) language. In the first case the adaptation is performed as an ordinary transformation, where in the latter it is performed by a dedicated tool.

4.3 Adaptation of service enactment due to design alterations

The most obvious adaptation case is when a service execution needs to be updated due to design alterations. In this case, the initial design models of organisation, coordination and service has been adapted with new structures/roles, coordination patterns and service functionalities. In our scenario, this is because the designer due to some external circumstances has re-evaluated the severity of the incident from *Grip-Level* 0 to 1. In the opposite direction now, the changes in models would propagate events which may cause a sequence of inner adaptations. Finally, from the adapted models an updated service enactment will be generated.

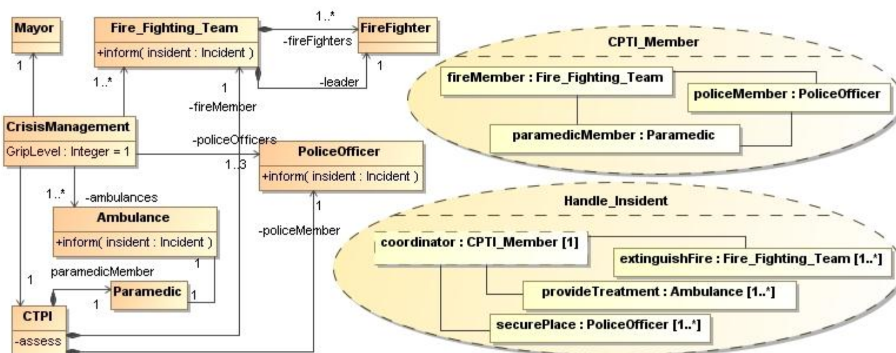

Fig. 5. After design-adaptation Organisation models at GRIP 1

Organisation: In GRIP-1, a local coordination team (CTPI) is now set up to supervise the operations and a *Mayor* entity is introduced. The CPTI team is composed of the heads of active services, such as *Fire_Fighting_Team* and *Police-Officer* and *Paramedic*. Additional forces have been reserved, so cardinality has changed to [1..n]. A new collaboration *CTPI_Member* defines the additional roles of *fireMember, policeMember* and *paramedicMember,* which can be played by existing handling members such as a *PoliceOfficer*. Finally, within the *Handle_Incident* participation, all police, ambulance and fire units on the ground communicate through a *CPTI_Member*, playing the role of a *coordinator*.

Coordination: At GRIP 1 the *Mayor* does not play an active role (there are no out-coming interactions), however he/she might get *informed by* the CTPI members. How information is exchanged and shared among members has also changed from a network to a hierarchical structure. Now every incident handler has an obligation to report directly to CTPI members. CTPI has also the right (permission) to delegate tasks to other non-CTPI members, whereas other non-CTPI members have the obligation (implement the interface which is accessible only to *CPTI_Members*) to perform the tasks delegated to them.

Services: At GRIP 1 two additional services *MayorService* and *CPTIService* are introduced. Previous services have also been altered in order to be consistent with the new coordination patterns. As a result, a *CPTIService* utilises the corresponding provided interfaces of *FireService*, *PoliceService* and *AmbulanceService* to delegate tasks as well as the *MayorService* to provide incident reports.

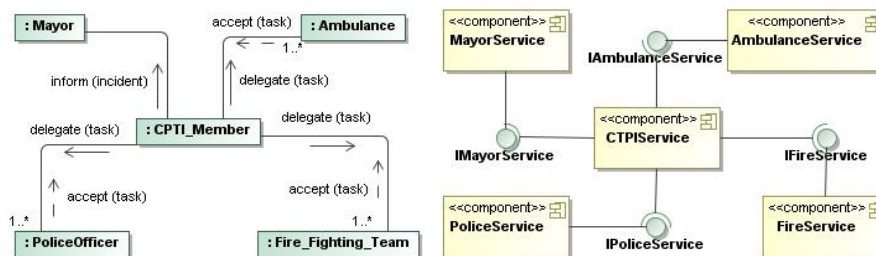

Fig. 6. After design-adaptation Coordination (left) and Service (right) models at GRIP 1

5 Other Related Work & Discussion

Another quite related approach to the concept of run-time models is that of executable UML [11]. Executable UML is based on rich diagrams that can produce executable models, which can then be translated directly to code. In this case a virtual machine interprets the UML models without any intermediate code generation step involved.

In our case the run-time models are represented by ordinary UML diagrams capturing the organisation and coordination of dynamic instances of an ALIVE scenario. Consistency among the service enactment (execution) and design models is maintained by the specification of significant events that are bound with specific state changes. Thus our approach does not provide a full bidirectional consistency among real execution (states) and dynamic models as the overhead is significant. Further, our ALIVE models are not executable; however they generate artefacts which can be

executed via a transformation process. Specific markings are also used to identify the significant states requiring monitoring and operations that may trigger an adaptation process.

6 Conclusions

Providing mechanisms facilitating the dynamic adaptation of design models and run-time executions is an important property for systems that need to reflect the environmental and design changes. In this paper we have proposed a mutual monitoring mechanism for maintaining adaptations among design models and service enactment. The run-time adaptations are performed automatically, triggered by significant events, directed by adaptation patterns described at design-time and implemented via model transformations. In addition, we have shown how the multi-layers of model abstractions add significant complexity in the adaptation process, which also needs to be supported by the mechanism.

References

1. Jeffrey, O.K. and M.C. David, *The Vision of Autonomic Computing.* Computer, 2003. 36(1): p. 41-50.
2. ALIVE. *Coordination, Organisation and Model Driven Approaches for Dynamic, Flexible, Robust Software and Services Engineering*. European Commission Framework 7 ICT Project 2008; Available from: http://www.ist-alive.eu/.
3. Clarke, S., A. Staikopoulos, S. Saudrais, J. Vázquez-Salceda, V. Dignum, W. Vasconcelos, J. Paget, L. Ceccaroni, T. Quillinan, and C. Reed, *ALIVE: A Model Driven approach for the Coordination and Organisation for Services Engineering*, in *to appear on International Conference on Model Driven Engineering Languages and Systems (MODELS 08) Research Project Symposium*. 2008: Toulouse, France.
4. France, R. and B. Rumpe, *Model-driven Development of Complex Software: A Research Roadmap*, in *2007 Future of Software Engineering*. 2007, IEEE Computer Society.
5. Wooldridge, M. and N. Jennings, *Intelligent Agents: Theory and Practice.* In The Knowledge Engineering Review, 1995. 10(2): p. 115-152.
6. Splunter, S.v., T. Quillinan, K. Nieuwenhuis, and N. Wijngaards, *Alive Project: THALES Usecase - Crisis Management.* Technical Report ALIVE Project, 2008.
7. Aldewereld, H., F. Dignum, L. Penserini, and V. Dignum, *Norm Dynamics in Adaptive Organisations.* 3rd International Workshop on Normative Multiagent Systems (NorMAS 2008), 2008.
8. Bézivin, J., S. Hammoudi, D. Lopes, and F. Jouault, *An Experiment in Mapping Web Services to Implementation Platforms*, in *Technical report: 04.01*. 2004, LINA, University of Nantes: Nantes, France.
9. Bordbar, B. and A. Staikopoulos, *On Behavioural Model Transformation in Web Services.* Conceptual Modelling for Advanced Application Domain (eCOMO), 2004. LNCS 3289: p. 667-678.
10. Foundation, T.A.S. *Apache ODE (Orchestration Director Engine)* 2008; Available from: http://ode.apache.org/.
11. Stephen, J.M. and B. Marc, *Executable UML: A Foundation for Model-Driven Architectures*. 2002: Addison-Wesley Longman Publishing Co. 368.

Modeling and Validating Dynamic Adaptation[1]

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

[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 can be executed.

1 Introduction

In [6], 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 [2] 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 does 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 [6] 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.

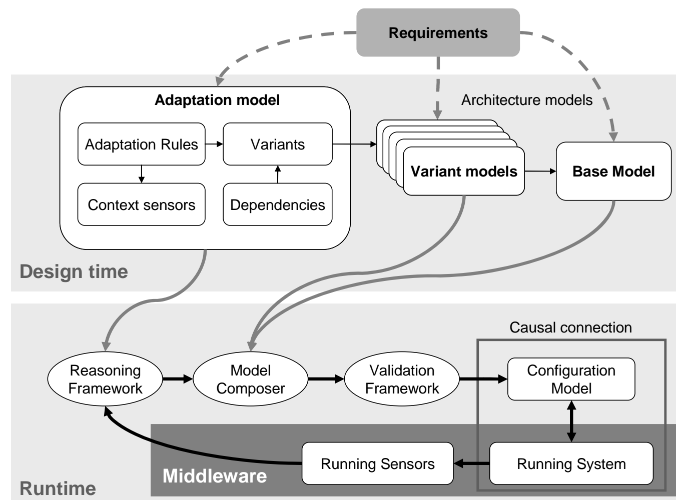

Fig. 1. Overview of the proposed approach

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 are presented in [4]. 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 [4]. 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 [6]. 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 includes 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.

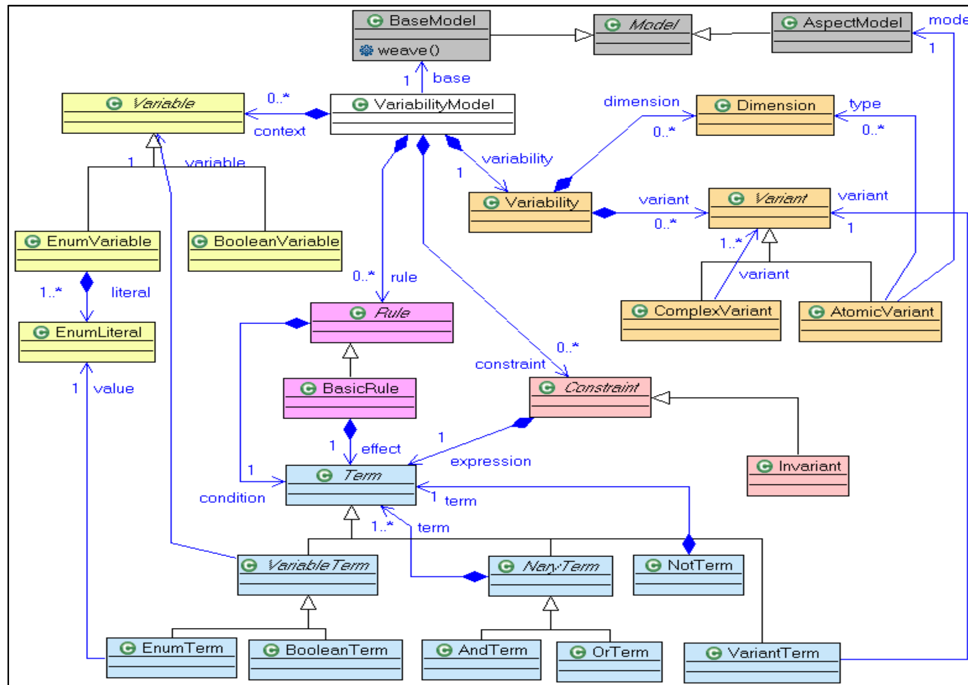

Fig. 2. Meta-model for variability and adaptation

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 6 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 7 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.

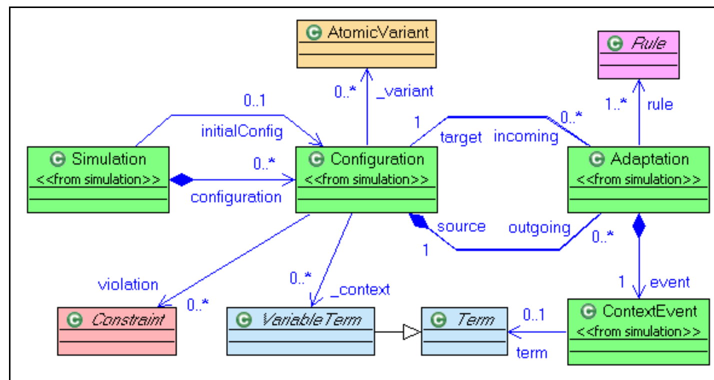

Fig. 7. Simulation model

Based on this simulation model, a prototype simulator has been implemented using the Kermeta platform [8]. 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 evaluates 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. Figure 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 in Figure 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 verify invariants on the system.

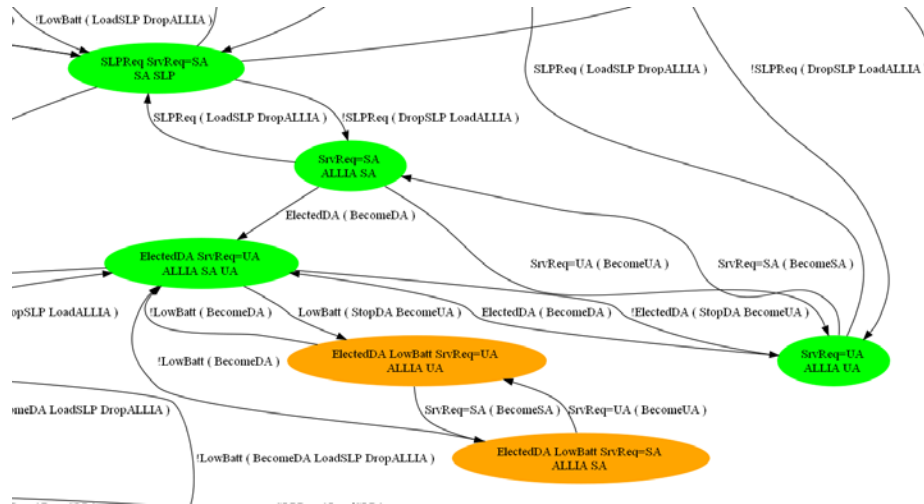

Fig. 8. Excerpt of the simulation graph for the SDA

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 computes 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), if 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). 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 [3] 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's 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 [5]. 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 [5, 7].

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 of 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 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] N. Bencomo, R. France, and G. 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.

[3] 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.

[4] C.A. Flores-Cortés, G. Blair, and P. Grace. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad-hoc Environments. *IEEE Distributed Systems Online*, 8, 2007.

[5] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, and J-M. Jézéquel. Introducing variability into aspect-oriented modeling approaches. *MoDELS'07: 10th International Conference on Model Driven Engineering Languages and Systems*, USA, October 2007.

[6] B. Morin, F. Fleurey, N. Bencomo, J-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. *MODELS'08: 11th International Conference on Model Driven Engineering Languages and Systems*, France, 2008.

[7] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J-M. Jézéquel. Managing variability complexity in aspect-oriented modeling. *MODELS'08: 11th International Conference on Model Driven Engineering Languages and Systems*, France, 2008.

[8] P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. *MoDELS'05: 8th International Conference on Model Driven Engineering Languages and Systems*, Jamaica, 2005. See http://www.kermeta.org/.

[9] J. Zhang and B. Cheng. Model-based development of dynamically adaptive software. ICSE'06: 28th International Conference on Software Engineering, China, 2006.

Runtime Models for Self-Adaptation in the Ambient Assisted Living Domain

Daniel Schneider, Martin Becker

Fraunhofer IESE, Fraunhofer Platz 1,
67663 Kaiserslautern, Germany
{Daniel.Schneider, Martin.Becker}@iese.fraunhofer.de

Abstract. Ambient Assisted Living systems (AAL) must fulfill several challenging requirements as, for instance, the ability to render their services at a quality level that is high enough to enable an independent living. This requires a sound understanding of the current situation of the users, their environment, and, the availability of required resources. Further, AAL systems need the ability to adapt and extend the system behavior, as the demands for living assistance substantially differs between different individuals (differential aging) and changes during a person's life. Both are requirements for a sound adaptation support at runtime that require adequate models.
In this paper we identify stereotypical adaptation scenarios in the AAL domain, elaborate on components and their respective models to support the adaptation scenarios, and discuss the evolution of these models.

Keywords: Self-Adaptation, Ambient Assisted Living, Context Management, Adaptation Management, Configuration Management

1 Introduction

Driven by demographical and societal changes in most industrialized countries, the development of Ambient Assisted Living (AAL) systems seems to be a promising answer to the question of how to enable people with specific needs, e.g. elderly or disabled people, to live longer independent lives in their familiar residential environments [1]. Typical services comprise assistance in daily routine and emergency treatment services. In order to succeed with this, AAL systems must meet several challenging requirements. Among these are (i) the ability to render their services at a quality level that is high enough to enable an independent living, which often requires a sound understanding of the current situation of the users and their environment and the availability of required resources, and (ii) the ability to adapt and extend the system behavior, as the demands for living assistance substantially differs between different individuals (differential aging) and changes during a person's life. Due to financial reasons, AAL systems will comprise in most cases just that hardware components and resources that are necessary to meet the current assisting demands. We call this the "just enough principle". Obviously, the situation awareness and

adaptability while tackling with resource constraints are among the stereotypical characteristic of AAL systems.

In the context of the BelAmI [2] project solutions and engineering approaches for runtime-adaptable solutions have been investigated in an application-driven manner. Among the addressed issues was to identify stereotypical adaptation scenarios, as well as to provide a system architecture, respective models and components that meet these requirements.

The investigation of the state-of-the-art on adaptable and adaptive systems revealed a substantial amount of approaches and solutions that already exist. However their applicability in the AAL context was for various reasons (e.g. impact on performance and resource consumption, timeliness, flexibility) not that clear. In addition, the relevance of the adaptation scenarios addressed by the approaches was often difficult to assess. This complicates the application of those approaches and solutions in the AAL domain substantially.

In order to pave the way for solutions that meet the adaptation demands of the AAL domain, the paper (i) identifies stereotypical adaptation scenarios in the AAL domain, (ii) elaborates on components and their respective models to support the adaptation scenarios, and, (iii) discusses the evolution of these models.

The paper is structured as follows: Section 2 describes the AAL domain and identifies typical adaptation scenarios therein. By means of these scenarios we deduce architectural entities and their respective runtime models in section 3, 4 and 5 and discuss the evolution of these models. Section 6 gives a brief overview on related work. We conclude in section 7 and give a short outlook on future work.

2 AAL Domain and Adaptation Scenarios

Ambient Assisted Living (AAL) [1] denotes concepts, products, and services that interlink and improve new technologies and social systems, with the aim of enhancing the quality of life for all people during all stages of their lives. AAL could therefore be translated best as "intelligent systems of assistance for a better and safer life" [3]. The potential range of services belonging to the AAL domain is huge. It encompasses any assistive service that facilitates daily life. The classification scheme in [5] structures this domain into six stereotypical subdomains with clearly separated responsibilities. As classification parameters, are used: (i) location where the assisted living service is rendered (row), and (ii) assistance types (columns):

	Emergency Treatment Services	Autonomy Enhancement Services	Comfort Services
Indoor Assistance	prediction detection prevention	drinking eating cleaning cooking dressing medication	logistical services services for finding things Infotainment services
Outdoor Assistance	prediction detection prevention	shopping assistance travel assistance banking assistance	transportation services orientation services

Figure 1 A classification scheme for the AAL services [5]

Within the BelAmI [2] project our primary interest is on the indoor emergency treatment services as they form the kernel of any AAL system that shall enable people to live an independent live alone at home.

In this paper we will use the following system fragment as running example: The system automatically senses the location and activities of the user to detect actual emergencies, e.g. sudden falls, or noticeable trends that could lead to a critical situation, e.g. the sudden increase of toileting activities or the decrease of drinking activities. In case of "suspicious" situations, the system checks with the user the real situation, and informs adequate assistance personnel if required. The system consists of a location device, a cup that senses drinking activities, a concentrator device that renders the emergency services, and a series of interaction devices, as pushbuttons, and several optional information displays. The concentrator is connected to the internet and allows a remote management of the system by a service provider.

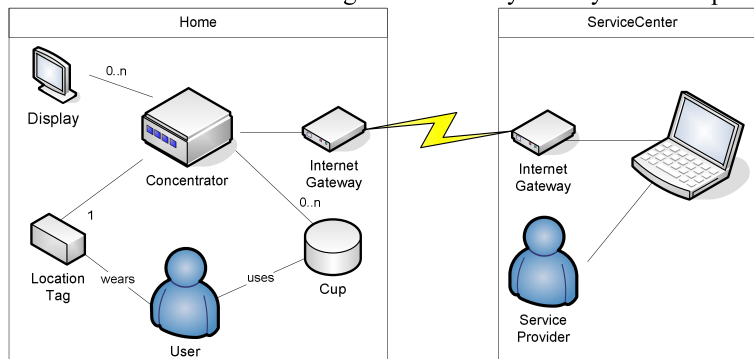

Figure 2 Example System

Within the system, we can identify the following types of adaptation scenarios

S1:Local Adaptation: Here we refer to adaptation scenarios where the adaptation decision is rendered locally, within the component that is to be adapted. Consider the following example: The localization device decides to decrease (downscale) it's sampling rate to save energy, if its location has not changed much in the past. Conversely, it will increase (upscale) its sampling rate if its location changed. The interesting point in this scenario is that the device decides locally its adaptation, and that the adapted parameters could be continuous.

S2:Remote Adaptation: In contrast to local adaptation, this type of adaptation scenario refers to adaptation decisions made by an external (dedicated) component. Consider the following as an example scenario: A simple drinking reminder is integrated into the cup (beeper), and thus enables the cup to render a complete drinking reminding service on its own. However, if the device is brought into an environment where a better reminder service is available, the reminder on the cup is deactivated and the better one is used. This operation is to be reverted if the cup and the concentrator are disconnected, or if the information display is deactivated. The characteristic problem in this scenario is that the adjustment takes place through the interaction of several distributed devices, and that only discrete features, activation or deactivation, are affected.

S3:Conflict negotiation (local adaptation goals vs. global adaptation goals). If both, local and global adaptations, are supported there might be conflicting adaptation decisions. The following scenario exemplifies this: The location device tries to save energy by switching into a low-power mode after a certain period of inactivity. In an emergency situation, an external agent must be able to prevent this default behavior. The typical trait of this scenario is that a device has to react reliably according to conflicting commands.

S4:Set point adaptation: In the set point adaptation scenario, a default set point for triggering a certain event is adjusted in a specific context, and the machine is perceived to adapt its parameter by learning. For example, the set point for triggering an alarm if a person has an abnormally high pulse is raised if that person executed a temporary exhaustive action. The machine registers that these two events have happened together and adapts accordingly, so that it is able to infer abnormal changes when a similar event happens again in the future.

S5:Manual adjustment: In the manual adjustment scenario, maintenance personnel are able to adjust the current composition, adaptation parameters and also adaptation rules. A further aspect of this scenario is the exchange of components, i.e. adding new components and removing old components. The characteristic point in the manual adjustment scenario is the manual intervention with either maintenance or diagnostic purposes.

In the following sections we elaborate on the components and their respective models we have added to our system architecture in order to support these adaptation scenarios.

3 Configuration

We have started with the simplest adaptation scenario, the manual adjustment S5. Here the adaptation is conducted by maintenance personnel at runtime. They are supported by the system to change the configuration of the system and its components, e.g. for the cup the volume of the beeper or the mode (normal, energy_saving) can be configured. With configuration we refer to a point in the configuration space. The configuration space is spanned by the supported configuration parameters (dimensions) and can be constrained by constraints. Different component configurations can result in different "internal" behaviors as well as in different connection topologies (compositions). A configuration can comprise one or many components of the system.

To realize dynamic (re)configuration we followed the ComponentConfigurator pattern of Buschman et al [6]. The aim of this design pattern is to "*allow an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application.*"

According to the pattern, an interface should be defined which allows to configure the implementer. This interface evidently must be implemented by every component that should be configurable. Furthermore, a Configurator is required to coordinate the (re)configuration of components, being especially of interest, if several components have to be configured in a coherent way. The Configurator implements a mechanism

to interpret and execute scripts specifying the configuration, which is to be instantiated. We correspondingly introduced a Configurator component as an interface which is to be implemented by configurable components (cf. Figure 3).

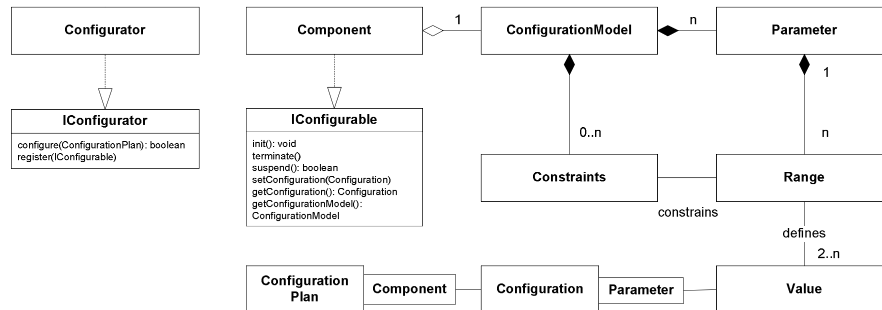

Figure 3 Configuration Support

The responsibilities of the Configurator component (cf. Figure 4) are threefold:

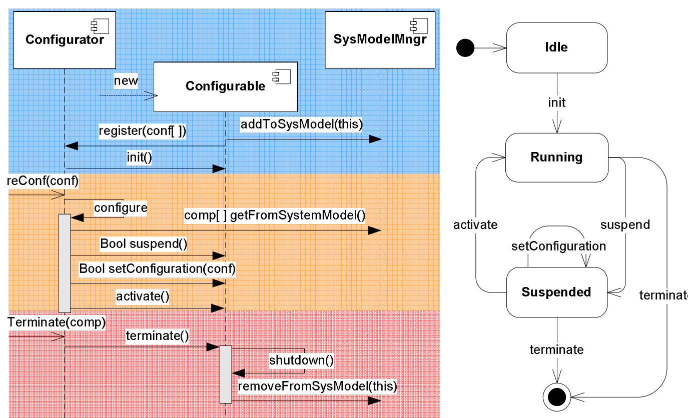

Figure 4 Configurator Responsibilities and Configurable Lifecycle

Component initialization: When a new component is loaded into the system it introduces itself to the SystemModelManager (SMM) and to the Configurator. The SMM maintains a model of the current system comprising the components and their current configurations. The model hence covers both, current component structure and parameters. After the registration phase, the Configurator configures the component into a predefined configuration.

Component (re-)configuration: The Configurator's main task is to manage the transition from a current configuration to a target configuration at runtime. In the manual adjustment scenario, such target configurations are provided by a maintenance engineer. To build a configuration, the maintenance engineer relies on information about the available components and their required and provided properties.

If other components are required to instantiate a certain configuration, the Configurator tries to get them from the SMM. For each of these components, the

Configurator then suspends the component before he applies the new configuration. If the instantiation of the new configuration was successful for all participating components the Configurator reactivates all suspended components.

Component termination: When a component is no longer needed the Configurator is responsible for terminating it in a controlled way (i.e. conduct reconfigurations of connected components when necessary).

4 Context-Awareness

Any sophisticated assistance function in the AAL domain and beyond requires a sound understanding of the current situation in order to plan and execute necessary actions. Self-adaptation falls into that class of functions, as the system adapts itself and thus relieves the system maintainers of this task. Situation awareness can be achieved by a rather simple sensor fusion, e.g. in the cup in our example system, or through a sophisticated fusion of information from different information channels that is known as context awareness in recent years and forms one of the most important ingredients for achieving the Ambient Intelligence **Fehler! Verweisquelle konnte nicht gefunden werden.** or Ubiquitous Computing paradigm [7]. Context awareness comprises three main functionalities: context sensing, context fusion (aka. context interpretation) and context management. Intuitively, many people perceive 'context' as aspects from the users' environment like location and temperature. Despite this common notion, it is hard to define context precisely. We adopt the general definition proposed in [8]: "*...Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.*" Context-awareness denotes the use of contextual information in computer system functionality.

In order to realize all above listed scenarios that comprise self-adaptation, we added a context management system to the architecture as illustrated in Figure 5. It supports the various system components with access to context information in a device-independent manner. The system provides a push (asynchronous) as well as a pull (synchronous) modus to access information. The context information is managed by the ContextManager in the system. It encapsulates the context distribution strategy in the distributed system and provides the various components with a n:m connector for context information distribution. The managed information and its interrelations are defined by the ContextModel, which is a meta-model. To make use of the context, any application component can register itself with the ContextManager as IContextProvider or IContextSubscriber. Basic context information is provided by sensors (IContextProvider) that provide data on various parameters in the environment, e.g. location of user or objects, and the system itself, e.g. system mode. Components that want to be notified on changes in the context information can register themselves as IContextSubscriber. Components that produce new context information based on basic context information are called IContextAggregators (they are subscribers and providers as well). They can be considered as a kind of logical sensor. The ContextManager also assures the persistence of context information that

should be stored for later retrieval. There are various ways to represent context information within the systems. Familiar approaches are *Key-value pairs* (uses the key to refer to the variables and the value of the variable holding the actual context data), *tagged encoding* (the context data is modelled by semi-structured tags, and attributes, e.g. in XML), *object-oriented models* (the context data is embedded in object states, and objects provide methods to access and modify the states), and, *logic-based models* (context data are expressed as facts in a rule-based system).

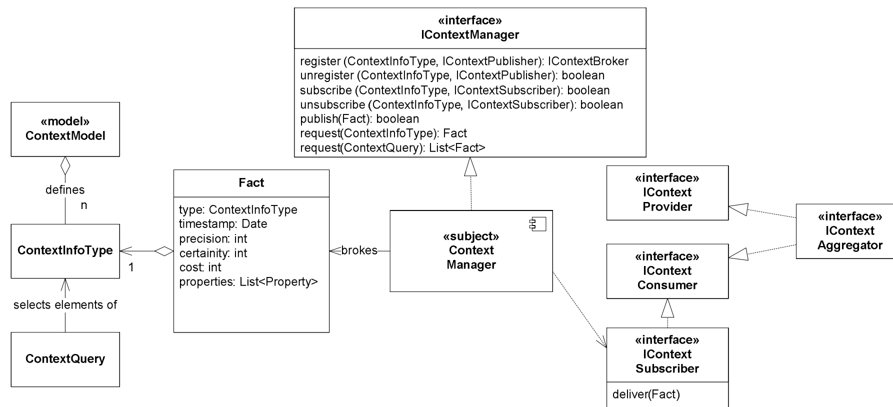

Figure 5 Context Management

We have followed the Tagged encoding approach, which allows us to realize a rather lightweight context management approach and provides us with sufficient classification facilities. The ContextModel defines the context information that can be queried and the quality of the context information. It consists of a list of context information topics with the respective attributes. The topics are structured hierarchically, which has been inspired by the OSGi event mechanism. This allows an efficient registration and broking of a group (subtree) of context information items via wildcards. Currently, we are considering migrating to an OWL-based [9] classification scheme, to provide more flexility in the type structure. The ContextModel is also a suitable place to specify, which context information is transient or should be stored for later retrieval. The ContextModel only needs to be evolved if new context information should be processed by the system, e.g. when a new sensing device is added or an improved ContextAggregator is added.

All context information within the system is represented as facts as depicted in Figure 5. Facts have a general, simple data structure which provides information about the type, time, precision, certainty, and cost of the fact, as well as a set of properties. Also the quality of the fact is of interest, as the quality of the context information can change over time.

Based on the ContextManager, components that adapt themselves in a situation aware way already can be realized, e.g. S1, S4. However, it was our goal to clearly separate adaptation logic from functionality in components, in order to make the adaptation logic explicit and to support global optimizations. To this end we

introduced a third component: the AdaptationManager that is described in the next section.

5 Adaptation Manager

In order to address the scenarios S2 and S3 the Configurator and ContextManager are not sufficient. In these scenarios there is no human in the loop and hence the system must be able to take over the maintenance engineer's tasks. More precisely, the system must analyze the current situation and plan corresponding changes when necessary. These changes can then be executed by the Configurator as described in section 3. To this end, we introduce a further platform component: the AdaptationManager (cf. Figure 6). The AdaptationManager consumes the context in order to evaluate the current situation and to decide upon possible system changes. To this end, all adaptable components have registered their AdaptationModel with the AdaptationManager and ConfigurationModel with the Configurator. Whenever the AdaptationManager identifies a necessary change, it plans the change (ConfigurationPlan) and sends it to the Configurator. The Configurator then takes care for the execution of the configuration. As the Configurator, the AdaptationManager needs to know which system variants exist. However, the information contained in the ConfigurationModel does not suffice. Deeper knowledge is required to identify situations in which configuration should take place and to define which configuration to take in which situation. Several approaches are reasonable to this end:
Rule-based approaches are widely spread in the domain of embedded systems. One reason is that such systems heavily rely on light-weight approaches due to the inherent scarcity of resources (e.g. processing time) and must be deterministic. The rule sets are to be defined at design time and usually have a "Event-Condition-Action" (ECA) form.

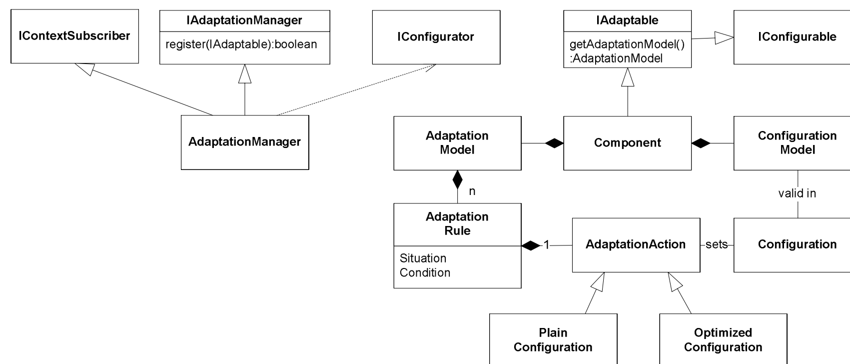

Figure 6 Adaptation Management

Goal-based approaches equip the system with goal evaluation functions. Neglecting available optimization strategies, the brute force approach would determine and evaluate all valid system configurations at runtime and choose that

variant that meets best the given goals. Thus, goal based adaptation is quite flexible and it is likely that optimal configurations are identified at runtime.

We follow a hybrid approach that allows the combination of both approaches depending on the actual adaptation needs. Overall we specify the adaptation behavior with AdaptationRules. In their action part, we distinguish between PlainConfiguration, which directly sets a predefined configuration or OptimizedConfiguration, which relies on an utility function to select an appropriate configuration. An example that illustrates the application of the latter is scenario S2, where the best output mechanism is used, depending on the current location of the user and the devices. Here the utility function maps the distance between the user and device to the utility value.

6 Related Work

In recent years numerous approaches emerged in the areas of adaptive systems. Many of them have been motivated by the upcoming paradigms of ubiquitous computing [7] and Ambient Intelligence (AmI). Prominent examples are Robocop, Space4U, and Trust4All. Robocop and Space4U extend the KOALA component model [10] with respect to different component views with corresponding models [11] and dynamic binding of components. Trust4All introduces the notion of trustworthiness and provides means to preserve a certain system level of dependability and security at runtime [12]. With a particular focus on adaptivity, MADAM and the follow-up MUSIC are closely related to our work. These approaches enable runtime adaptation by exploiting architecture models and generic middleware [13][14]. MUSIC builds upon the results of MADAM and introduces different extensions and optimizations [15].

Apart from the largely middleware centric work in the AmI domain there exist recent results in the area of model driven engineering of adaptive systems. Cheng et al. introduced a method for constructing and verifying adaptation models using Petri nets [16]. An interesting approach using software product lines and model driven techniques [17] as well as corresponding tool support [18] to develop adaptive systems is proposed by Bencomo et al. Their work will be of particular interest for our future work.

7 Conclusion and Outlook

Situation awareness and adaptability while tackling with resource constraints are among the stereotypical characteristic of AAL systems. Over the last years, a substantial amount of approaches and solutions emerged in this context. However, their applicability in the AAL context was for various reasons not that clear. In addition, the relevance of the adaptation scenarios addressed by the approaches was often unclear. Therefore we have elaborated on components and their respective models to support typical adaptation scenarios in the AAL domain and have raised some issues with regard to the evolution of these models.

Currently we are implementing our concepts in our ambient assisted living lab. The configurator, adaptation manager and context manager components have been realized as OSGi service bundles. As of now, the different adaptation scenarios can be supported. However, we still need to do some consolidation and hence corresponding refinement and validation of the presented mechanisms is our next step.
The main goal of our future work is to come up with a solution for variability management that can be continuously applied in the lifecycle of a software intensive product (family), ranging from development time to runtime.

Acknowledgements
Part of this work has been funded by the Federal Ministry of Education and Research (BMBF) and the Ministry of Science, Continuing Education, Research and Culture of the state of Rhineland-Palatinate in the context of the BelAmI [2] and AmbiComp projects [3].

References

[1] Ambient Assisted Living Joint Program, http://www.aal-europe.eu, last visited 19.09.2008.
[2] BelAmI, Bilateral German-Hungarian Collaboration Project on Ambient Intelligence, http://www.belami-project.org, last visited 19.09.2008.
[3] http://www.ambicomp.org/ (german site), last visited 19.09.2008.
[4] mst news: "Ambient Assisted Living", mstnews 06/07, http://www.mstnews.de/past-issues, last visited 03.03.2008.
[5] J. Nehmer, A. Karshmer, M. Becker, and R. Lamm, “Living Assistance Systems – An Ambient Intelligence Approach”, International Conference on Software Engineering (ICSE), 2006.
[6] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, “Pattern-Oriented Software Architecture.”, Wiley, 2000.
[7] M. Weiser, “The Computer for the Twenty-First Century”, Scientific American, p. 94-104, September 1991.
[8] A. K Dey, D.Salber, G. D. Abowd, “A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications”, Human-Computer Interaction 16, pp. 97-166, 2001.
[9] OWL Report, http://www.w3.org/TR/owl-features/
[10] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, “The Koala component model for consumer electronics software”, IEEE Computer, vol.33, no.3, pp.78-85, Mar 2000.
[11] J. Muskens, M. Chaudron, “Integrity Management in Component Based Systems”, Proc. of 30th EUROMICRO Conference (EUROMICRO'04), pp. 611-619, 2004.
[12] G. Lenzini, A. Tokmakoff, and J. Muskens, “Managing Trustworthiness in Component-based Embedded Systems”, Electron. Notes Theor. Comput. Sci. 179, 143-155, Jul. 2007.
[13] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, Eli Gjorven, "Using Architecture Models for Runtime Adaptability," IEEE Software, vol. 23, no. 2, pp. 62-70, 2006.
[14] S. Hallsteinsen, E. Stav, A. Solberg, J. Floch, "Using product line techniques to build adaptive systems", 10th International Software Product Line Conference, pp. 21-24, 2006.
[15] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E Stav, “Composing Components and Services using a Planning-based Adaptation Middleware”, Proc. of 7th Intl. Symp. on Software Composition (SC'08), Springer LNCS, pp. 16, Budapest, Hungary, 2008.
[16] J. Zhang, and B. H. Cheng, “Model-based development of dynamically adaptive software”, 28th International Conference on Software Engineering, Shanghai, China, 2006.
[17] N. Bencomo, P. Sawyer, G. Blair, and P. Grace, "Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems", 2nd International Workshop on Dynamic Software Product Lines, Limerick, Ireland, 2008.
[18] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair, "Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems", 30th International Conference on Software Engineering, Formal demos at ICSE 2008, Leipzig, Germany, 2008.

FAME—A Polyglot Library for Metamodeling at Runtime

Adrian Kuhn and Toon Verwaest

Software Composition Group,
University of Berne, Switzerland
http://scg.iam.unibe.ch

Abstract. Tomorrow's eternal software system will co-evolve with their context: their metamodels must adapt at runtime to ever-changing external requirements. In this paper we present FAME, a polyglot library that keeps metamodels accessible and adaptable at runtime. Special care is taken to establish causal connection between fame-classes and host-classes. As some host-languages offer limited reflection features only, not all implementations feature the same degree of causal connection. We present and discuss three scenarios: 1) full causal connection, 2) no causal connection, and 3) emulated causal connection. Of which, both Scenario 1 and 3 are suitable to deploy fully metamodel-driven applications.

Keywords: causal connection, eternal systems, metamodeling at runtime.

1 Why Metamodeling at Runtime?

Metamodeling is at the core of many web- and business applications. It is the means through which meta-information about the domain of a system is represented. Metamodeling is traditionally limited to the static (*i.e.* design time) representation of meta-information. Most often, only business data is extendible and editable at runtime, whereas any change to the business model requires a re-design of the system, often involving major engineering effort. For example, an application may allow users to edit the content of forms, but not their structure and workflow.

In the vision of eternal software [1], running systems are imagined to adapt to various, often unanticipated, context changes with little or no engineering effort. An eternal system must co-evolve with its context: as the business changes over time, the system is required to extend and adapt its metamodel. Therefore, we advocate to extend common systems with the ability to not only edit and extend the contained data but also the metamodel at runtime. To meet this requirement, the structures of meta-information represented in a metamodel are to be kept accessible *and adaptable* at runtime. As such, they can be used to change and extend the metamodel at runtime—that is *after* the system has been put in use.

This approach has been realized in a library called FAME, which provides a lightweight kernel to represent both models (*i.e.* data) and metamodels using the same structures

at runtime. FAME has initially been developed as the kernel of Moose [2], a highly-adaptable Smalltalk application used for research in software- and information visualization. Within the last year, the library has been ported (at varying stages of specification conformance) to: Squeak, Java, Python, and partially, C# and Ruby.

This paper presents a complete description of Fame's API and design, including its lightweight meta-metamodel (FM3) and text-based exchange format (MSE). Design decisions and implementation issues are discussed. In particular, as some languages offer limited reflective features only, not all implementations of Fame feature the same degree of causal connection. We present and discuss three scenarios: 1) full causal connection, 2) no causal connection, and 3) emulated causal connection, of which both Scenarios 1 and 3 are suitable to deploy fully metamodel-driven applications.

In general, the approach taken by Fame is related to previous work on runtime metamodeling by Costa et al [3], Jouault et al [4], and Clark et al [5]—please refer to our previous work [6] for comprehensive list of references and related work. Parts of Fame's implementation originate from Renggli's Magritte library [7].

The remainder of the paper is structured as follows. Section 2 starts with an overview of FAME and discuss afterwards the core abstractions: in Section 3 the meta-architecture, in Section 4 the FM3 meta-metamodel, in Section 5 the Fame API, and in Section 6 model serialization. In Section 7 we present and discuss three scenarios for causal connection between fame-classes and host-classes. Finally, Section 8 concludes.

2 Fame in a Nutshell

The purpose of FAME is to attach meta-information to the objects of a running system. Fame provides a uniform interface on both objects and their meta-information, that is, both are manipulated with the same API calls.

Since Fame is a polyglot library we have taken special care to ensure that all implementations offer the same core API. Fame has currently been ported to: Smalltalk, Java, Python, and partially, C# and Ruby. The Java library acts as the reference implementation, whereas the Smalltalk library offers the most additional features.

At the very heart of Fame is a *tower of models* with three layers [8]. Each layer contains elements which conform to the meta-information specified by the layer above. The bottom layer M1 contains objects of your running application, the middle layer M2 contains meta-information about your objects, and the top-most layer M3 contains meta-information about the meta-information of you objects. Thus, we refer to these layers as model, metamodel and meta-metamodel layer. Even though, at runtime, any layer is editable, it is common to populate the top-most layer with a static set of elements. This static set of elements is referred to as FM3, the FAME meta-metamodel.

Additionally, elements of any layer are serializable to text stream and back using the MSE exchange format. MSE is a generic file format to exchange any kind of objects, regardless of the metamodel. Thus, FAME-compliant applications can exchange both data (*i.e.* models) and metamodels by the same means.

The complete specification of both FM3 and MSE, as well as the sources of FAME are available at http://smallwiki.unibe.ch/fame under GPL license.

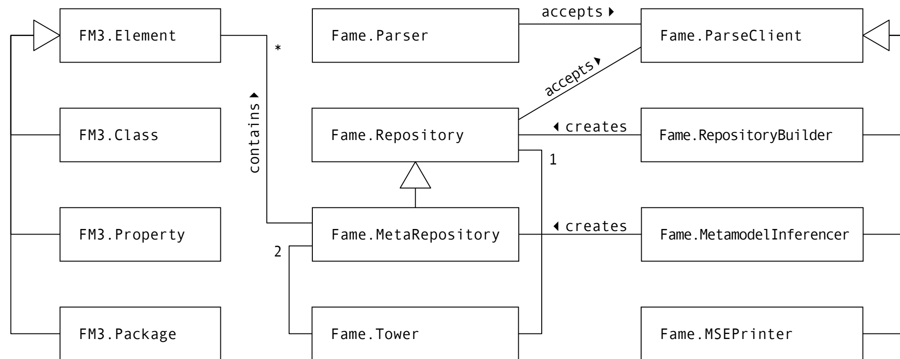

Fig. 1. Class diagram of FAME: including the FM3 meta-metamodel (left), Fame meta-architecture and API (center), and MSE serialization/streaming (right, top center).

Design and Core Abstractions of Fame

All FAME implementations are realized using the same design. Figure 1 presents a class diagram with the main host-classes[1], from left to right: Element, Class, Property and Package implement the FM3 meta-metamodel. Tower, Repository, and MetaRepository implement the meta-architecture and most of Fame's API. Parser, ParseClient and its subclasses are used for serialization and de-serialization of models.

3 Meta-Architecture —– Towers – Models – Elements

The major components of Fame's meta-architecture are towers, models, and elements. Towers contain models, models contain elements. There are two kinds of models: meta-repositories and repositories. The former contain FM3-compliant elements only, whereas the latter my contain any kind of host-language objects, typically domain objects. Often the terms model and repository are used interchangeably.

Each tower consists of three layers referred to as, from bottom to top: M1, M2, and M3. The elements on every layer must conform to the meta-information specified by the layer above. A typical setup is as follows: at the bottom layer a repository holding domain objects of the running application, at the middle layer a meta-repository holding the current domain model, and eventually, at the top-most layer, a meta-repository holding the self-described FM3 metamodel.

The tower is not implemented as a singleton because more than one application may run in the same object memory, each with its own tower. These towers may share the top layers. For example, if each open document of an application is represented by a concurrent tower, then the towers have different M1 layers but share the other layers.

[1] The term *class* is used by both object-oriented programming and the metamodeling paradigm, but with different meanings. Hence, we refer to the classes of the host-language as *host-classes* and to the classes of FM3-compliant metamodels as *fame-classes*.

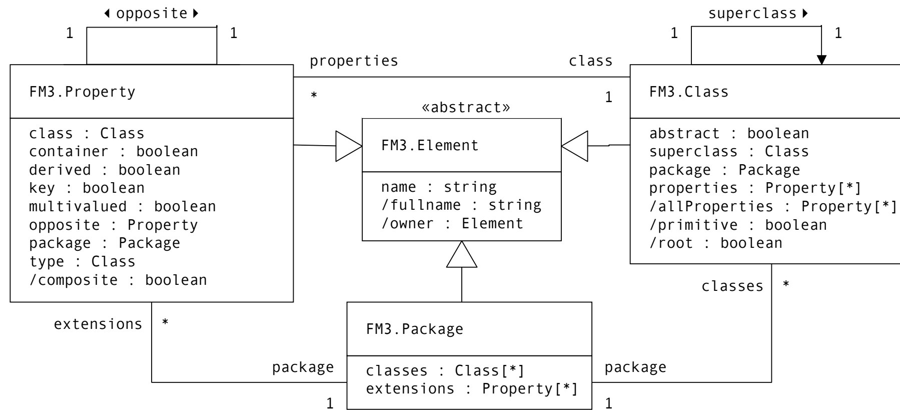

Fig. 2. Complete class diagram of FM3 meta-metamodel, including all properties and associations. Derived properties are marked with a leading slash character.

4 Meta-Metamodel —– Classes – Properties – Packages

All elements contained in a meta-repository must conform to FM3. These are all elements on the M2 and the M3 layer of a tower. Since the M3 layer is usually populated with elements that represent FM3, it conforms to itself.

C.A.R Hoare once said that "inside every large program there is a small program trying to get out," and in fact—FM3 is MOF 2.0 [8] reduced to the minimum. Meta, a precursor of Fame, has been used for many years as the kernel of Moose. Meta used MOF and later EMOF as its meta-metamodel. However, code reviews had shown that we only ever used a very small subset of EMOF, while many of the rarely used features had repeatedly been the cause of additional engineering effort.

Figure 2 presents the complete class diagram of FM3. There are three kinds of elements: packages, fame-classes, and properties. Packages contain classes and properties, classes contain properties. Packages can also directly contain properties which were not declared in the fame-class itself. In this case we say that those packages extend the fame-class and refer to the properties as *extensions*.

Extensions are one of the two additional FM3 features not present in EMOF. The other is unique keys. Both facilitate better metamodeling at runtime.

Extension properties are related to packaging and modularization of metamodels. Given package A and package B, package B can use extension properties to add properties to classes contained in A without tampering with the definition of A itself. This is useful for evolving applications. For example, for plugins or libraries that choose to extend the metamodel of an existing application.

Unique keys are related to the composition of associated elements. A fame-class must not have more than one keyed property. If an element with a keyed property k is contained in a container, the value of k must be unique with regard to the set of all values of k of all elements contained in the same container. This is useful for the improvement of the performance of runtime elements (discussed below in Section 7).

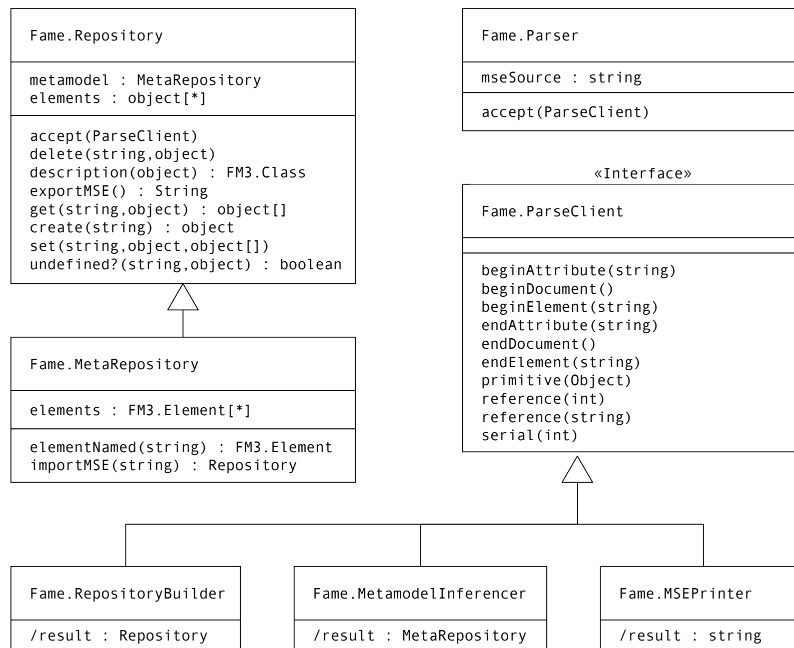

Fig. 3. Class diagram of the Fame API: (left) interfaces for runtime manipulation and elements and models, (right and bottom) interfaces for serialization and streaming.

The container can keep the composites in a host-dictionary rather than a host-collection, and thus provide access to composites by key in $O(1)$ rather than $O(n)$ time.

5 Fame API —– Create – Delete – Get – Set – Undefined

Since Fame is a polyglot library we have taken care to ensure that all implementations provide the same API. Fame offers functionality to

- manage one or more tower of models,
- manipulate elements and their meta-information,
- serialize and de-serialize (meta)models.

All elements (both objects and meta-information) are manipulated with the same API calls. This is possible since within a tower of models, objects and meta-information are “meta-described” in the same way. This allows for both models and metamodels to be serialized to the same exchange format.

Figure 3 presents the Fame API, from left to right/bottom: Repository and MetaRepository offer functionality to manipulate elements. Whereas Parser, ParseClient and its subclasses offer functionality to serialize and de-serialize complete (meta)models.

All elements have *attributes*. Each attribute has a name and optionally refers to another element. For each property of an element’s fame-class, there must be a corre-

sponding attribute with the same name as the property. The value of an attribute is either set (that is, referring to another element) or empty and thus undefined[2].

Depending on host-language and scenario, attributes are either realized as instance variables or using separate host-classes. In particular, bi-directional relations are often realized as separate host-classes so that manipulation of one end automatically updates the opposite end.

The Fame API offers these five operations on elements:

- *create(name)* creates a new element that conforms to the specified fame-class. All attributes are initially undefined (unless specified otherwise by their type).
- *delete(slot, element)* resets the value of an attribute to undefined.
- *get(slot, element)* either returns the value of the attribute or fails if the value is undefined.
- *set(slot,element,value)* sets the value of a slot to the specified value.
- *undefined?(slot,element)* returns true if the slot is undefined or false otherwise.

This set of operations is similar to the five methods of RESTful web-programming [9]. This is not a coincidence, as many of the actions on business objects can be expressed in terms of common operations on the object graph of a running application. The same applies to the manipulation of meta-information. Fame does not offer dedicated operations for the meta-information. Rather, meta-information is manipulated with the same operations, the only difference being that we are operating another layer of the model-tower. The operation *description(element)* is used to navigate from an element to its meta-information.

Depending on the host-language and scenario, these operations of the Fame API are offered on the Repository host-class only (as indicated by Figure 3) or on native objects as well. The latter is not supported by host-languages where host-classes are closed for extension, which is the case for Java, but not C# and the others.

6 Serialization and Streaming —– Parse – Infer – Print

Elements of any layer are serializable to text stream and back using the MSE exchange format. MSE is a generic file format to exchange any kind of objects, regardless of the metamodel. Thus, FAME-compliant applications can exchange both data (*i.e.* models) and metamodels by the same means.

Serialization of (meta)models is based on streaming. Figure 3 presents the main collaborators of streaming to the right and the bottom. There are two kinds of collaborators: producers and consumers. Producers are host-classes that offers an *accept(ParseClient)* method. Consumers are host-classes that implement the ParseClient interface.

Fame offers two default producers:

Parser parses an MSE-compliant text stream; and
Repository iterates over all contained elements.

[2] Please note, that undefined and nil are not the same: according to the FM3 specification, nil is a predefined instance of Object, whereas undefined describes the value of an attribute. The value of an attribute that holds a reference to nil is thus not undefined.

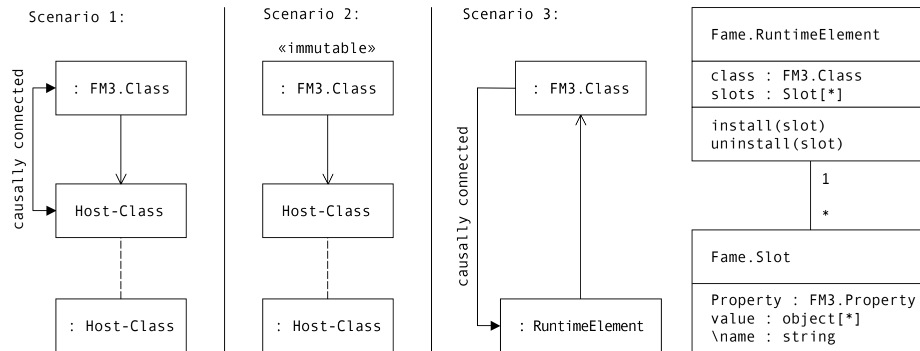

Fig. 4. Three scenarios of causal connection in Fame: (left), (center), (right).

It also offers default consumers:

RepositoryBuilder reads models given the stream of a model;
MetamodelInferencer infers the metamodel given the stream of a model; and
MSEPrinter serializes (meta)models to text streams.

The combination of inferencer and builder is able to import models for which the metamodel is not known or not available for some reason.

Models are exchanged between producers and consumers by means of a strict protocol. This communication is unidirectional. The producers fire a series of events to which the consumers listen. In case the protocol is not followed, the entire sequence is considered to be illegal. Each event corresponds to a method in the ParseClient interface. These methods must be called in following order (given as grammar rules):

$$\begin{aligned}
&\text{Sequence} &&= \text{Document}\ ? \\
&\text{Document} &&= \mathit{beginDocument}\ \text{Element}\ *\ \mathit{endDocument} \\
&\text{Element} &&= \mathit{beginElement}\ \mathit{serial}\ ?\ \text{Attribute}\ *\ \mathit{endElement} \\
&\text{Attribute} &&= \mathit{beginAttribute}\ \text{Value}\ *\ \mathit{endAttribute} \\
&\text{Value} &&= \text{Element}\ |\ \mathit{reference}\ |\ \mathit{primitive}
\end{aligned}$$

In fact, the above protocol corresponds to the grammar of the MSE exchange format[3] as specified in Appendix A. To avoid dependencies between streaming and meta-architecture, primitive parameters are only used in event signatures. Thus, serialized elements can be processed without setting up the entire meta-architecture first.

7 Causal Connection Between Fame-Classes and Host-Classes

Depending on host language and scenario, a causal connection between fame- and host-classes is maintained or not. Causal connection is required to update elements when

[3] The original meaning of MSE has been lost in the dust of history, however MSE is sometimes refered to as “Moose without objects” in reference to Fame’s origin as kernel of Moose.

their meta-information changes. There are three possible scenarios, as illustrated on Figure 4 from left to right:

- Each fame-class is associated with a corresponding host-class between which full causal connection is established. Changes to fame-classes are propagated to host-classes, and vice versa. The scenario is the default use case for languages that support hot swapping. Currently these are Smalltalk and Ruby, although it is also perfectly possible for Python.
- Each fame-class is associated with a corresponding host-class, but no causal connection is established. In fact, both classes are considered immutable and attempts to change fame-classes results in failure. Even though this scenario partly defies the purpose of Fame, it is default for Java, C# and Python.
- All model elements are instances of `RuntimeElement`, a dedicated host-class which emulates instances of fame-classes by using host-dictionaries. In this case fame-classes are not associated with host-classes, but rather directly referred to by the runtime elements. Causal connection between fame-classes and runtime elements is maintained. Changes to fame-classes are propagated and applied to all associated runtime elements. Currently this scenario is available for Smalltalk and Java.

Fame is a polyglot library and has been implemented (at varying stages of standard conformance) for many languages, each with its strengths and weaknesses. Even though all implementations offer the same core features, they differ in the degree to which causal connection between fame-classes and host-classes is maintained.

Due to missing hot swapping support by the host-language, the Java implementation offers the least degree of causal connection. The Smalltalk one offers the highest degree of causal connection.

In Smalltalk, FAME has been leveraged to the same level of abstraction and tool support as the host-language. For example, each manipulation that is applied to a meta-model is translated into a corresponding refactoring that is applied at runtime to the host-classes. Furthermore, both IDE and debugging tools have been extended to allow developers to inspect and manipulate the meta-information of any object at runtime.

Smalltalk development happens at runtime. You can think of its IDE as an advanced graphical REPL interface, and of editing source code as applying a sequence of runtime refactorings. It was only natural to extend these runtime development tools with support for metamodeling.

The degree of causal connection implies the degree to which a Fame implementation can bridge the gap between host-language and metamodeling at runtime. The above scenarios differ as follows

- Under Scenario 1 there is almost no gap between the host-language and metamodeling. Manipulating elements with host-language constructs or using the Fame API is equivalent, both have the same semantics.
- Under Scenario 3, RuntimeElement introduces a gap between host-language and metamodeling. All operations on runtime elements must use the Fame API.

For example, given element e in model m, there are two ways to set the value of a slot $zork$: either using host-language constructs `e.zork(value)`, or using the Fame API `m.set("zork",e,value)`. Scenario 3 supports the latter only.

Given a fully metamodel-driven application however, the difference between Scenario 1 on the one hand and Scenario 3 on the other hand is moot. Such an application will typically not require any code that bypasses the Fame API and can thus be realized under any of those two given scenarios of causal connection. Only Scenario 2 differs from the others as it is immutable. Any usage of the part of the Fame API which alters metamodels will result in failure.

A good example of a FAME-based application which is (almost) fully metamodel-driven is provided by Ducasse et al [6]. They present Moose, a research tool for software- and information visualization, that

- generates all UIs at runtime based on meta-information,
- exclusively uses the Fame API to interact with domain objects,
- exclusively uses the Fame API to interact with meta-information,
- exclusively uses Fame streaming to (de)serialize its data.

Moose is written in Smalltalk which employs Scenario 1. As long as Fame's API is never bypassed using host-language constructs, the same (or at least a similar) application could be written using Scenario 3 without loss of features or functionality. However, it remains open at which cost of runtime performance and development overhead.

8 Conclusion

FAME is a polyglot library for metamodeling at runtime. Fame attaches meta-information to the objects of a running application. The attached meta-infomration itself is described by another layer of meta-information, which is eventually described by itself. The API of Fame offers a common set of instructions to manipulate elements at any layer.

Special care is taken to integrate the meta-information as seamless as possible into the structures of the host-language. In particular, to establish causal connection between fame-classes and host-classes. As some hosts only offer limited reflection features, not all implementations of Fame offer the same degree of causal connection.

Due to missing hot swapping support by the host-language, of all FAME implementations, the Java one offers the least degree of causal connection. On the other hand, the Smalltalk one offers the highest degree of causal connection.

Fame supports three different scenarios of causal connection: 1) full causal connection, 2) no causal connection, and 3) emulated causal connection. All three are presented and discussed in the paper. We show that fully metamodel-driven applications can be written for both Scenario 1 and Scenario 3, as long as no host-language constructs are used to bypass Fame's metamodeling API.

Acknowledgments

We thank the anonymous reviewers for their corrections and interesting comments.

We gratefully acknowledge the financial support of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" and the Swiss National Science Foundation for the project "Analyzing, Capturing and Taming Software Change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

A Grammar of MSE Exchange Format

```
Root := Document ?
Document := OPEN ElementNode * CLOSE
ElementNode := OPEN NAME Serial ? AttributeNode * CLOSE
Serial := OPEN ID INTEGER CLOSE
AttributeNode := OPEN Name ValueNode * CLOSE
ValueNode := Primitive | Reference | ElementNode
Primitive := STRING | NUMBER | TRUE | FALSE | NIL
Reference := IntegerReference | NameReference
IntegerReference := OPEN REF INTEGER CLOSE
NameReference := OPEN REF NAME CLOSE

CLOSE := ")"
FALSE := "false"
ID := "id:"
INTEGER := digit +
NAME := letter ( letter | digit ) * ( "." letter ( letter | digit ) ) *
NUMBER := "-" ? digit + ( "." digit + ) ? ( "e" "-" ? digit + ) ?
OPEN := "("
REF := "ref:"
STRING := ( "'" [^'] * "'" ) +
TRUE := "true"
comment := "\"" [^"] * "\""
digit := [0-9]
letter := [a-zA-Z_]
whitespace := "\f" | "\n" | "\r" | "\s" | "\t"
```

All MSE text streams must use UTF-8 encoding.

References

1. Nierstrasz, O., Denker, M., Gîrba, T., Kuhn, A., Lienhard, A., Röthlisberger, D.: Self-aware, evolving eternal systems. Technical Report IAM-08-001, University of Berne, Institute of Applied Mathematics and Computer Sciences (2008)
2. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), New York NY, ACM Press (2005) 1–10 Invited paper.
3. Fabio Costa, Lucas Provensi, F.V.: Towards a more effective coupling of reflection and runtime metamodels for middleware. In: Workshop on Models at Runtime. (2006)
4. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Springer (2006) 171–185
5. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied metamodelling: A foundation for language driven development (2004)
6. Ducasse, S., Gîrba, T., Kuhn, A., Renggli, L.: Meta-environment and executable metalanguage using smalltalk: an experience report. Journal of Software and Systems Modeling (SOSYM) (2008) To appear.
7. Renggli, L., Ducasse, S., Kuhn, A.: Magritte — a meta-driven approach to empower developers and end users. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Model Driven Engineering Languages and Systems. Volume 4735 of LNCS., Springer (September 2007) 106–120
8. Group, O.M.: Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group (2004)
9. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (2007)

A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service

Audrey Occello[1], Anne-Marie Dery-Pinna[1], Michel Riveill[1]

I3S Laboratory, France
{occello, pinna, riveill}@polytech.unice.fr

Abstract. Dynamic software adaptations may lead application execution to unsafe states. Then, the detection of adaptation-related errors is needed. We propose a model-based detection in order to define a solution that can be used in multiple platforms. This paper is intended to show how a runtime model for monitoring adaptation safety may look like and how it may be concretized in order to interact with real applications.

Keywords. Runtime models, adaptation safety, monitoring, service.

1 Introduction

Software systems are becoming more complex as they are composed of distributed components using a variety of devices and platforms to deliver services to mobile end-users. Such complexity also increases the need of adapting these systems to feet the changing environment. As these adaptations involve changing the structure or the behavior of applications, they may lead application execution to unsafe states. For example, a functionality may be removed accidentally when removing a component or undesired cycles may be introduced in new interactions between components. The detection of adaptation-related errors is needed to control adaptation safety.

As well as Medicine can be preventive or curative, we can choose to repair or anticipate adaptation-related errors. Repairing an executing system is difficult because it assumes that we can resume to an earlier state of the application and often force to freeze a part or even the whole application execution. We believe that freezing the application while it is adapted in order to detect errors is not a good remedy for highly available applications. We prefer to monitor the applications and prevent unsafe adaptations to occur.

Models are usually used at design time to capture architectural decisions and to ensure a common understanding. Such information is generally not available at runtime because it is lost when the model elements are transformed into runtime artifacts. However, runtime models have been used for decades in the meta-programming research community [1] in the form of meta-object protocols [2] enabling systems to control themselves during their execution. As Muller and Barais said, "automatic or even self-adaptability of the running system may be achieved by taking decisions based on monitoring information captured by runtime models" [3]. This can be applied to our problem: a system can take the decision of accepting or discarding an adaptation request by monitoring the history of adaptations that have been already performed on it.

We adopt a Model Driven Engineering [4] approach in order to define a monitoring solution that can be used in multiple platforms and whose abstraction makes it possible to reason about the solution correctness as a one-time cost. In previous work, we detailed what kind of adaptation-related verifications are carried out [5], [6], how we modeled adaptation safety [7] and how we established the correctness of our modeling [8]. In this paper, we show how a runtime model for monitoring adaptation safety may look like and how it may be concretized in order to interact with real applications.

2 How does a runtime model for monitoring adaptation safety look like?

The main elements of our runtime model, called Satin are depicted in Figure 1. To illustrate these elements, we will take, as a running example in the next section, an application made of diaries.

2.1 Software entities to be monitored

Components

In Satin, a **component** represents any software instance (unit of execution) composing the application and that is capable of exhibiting its interface(s): object, component, service, etc. Only the information about the identity and implemented interfaces of such software instance is reified (we do not deal with the software instances' state).

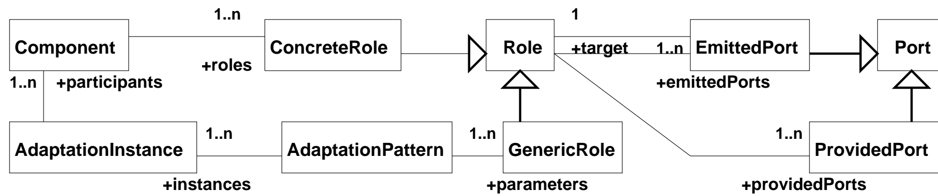

Fig. 1. Satin runtime model overview

Consider *lizDiary* and *johnDiary*, two diary instances of our running example, implementing a same interface with the add, remove, get and query meeting functionalities. At the Satin model level, *lizDiary* and *johnDiary* are two instances of the **Component** class.

Component roles
A Satin component is associated with **roles** (see the `ConcreteRole` class in Figure 1) which reify information about the software instance's implemented interfaces. Roles are composed of **ports** which are abstractions of operations provided by the component (see the `ProvidedPort` class in Figure 1) or required by one of the component adaptations (see the `EmittedPort` class in Figure 1). The interface implemented by the diary components is represented by the *BasicDiary* role. *BasicDiary* provided ports are: *addMeeting*, *removeMeeting*, *getMeeting*, *isFree* corresponding to functionalities that are present in the component interface at the platform level.

At any time, the roles of a component reflect its interactions with the environment. At the beginning, components' roles have only provided ports. When a component is adapted, its roles evolve and emitted ports are added to these roles. Roles are not shared between components. *lizDiary* and *johnDiary* have their own *BasicDiary* role and they evolve independently.

Adaptation patterns
In platforms, runtime adaptations can be structural and behavioral and consists of a set of elementary actions, which can express: 1) addition, withdrawal or replacement of components within an assembly, 2) addition and withdrawal of component ports, or 3) behavior modification of component ports.

In Satin, an **adaptation pattern** represents the unit of application and reuse of platform adaptations. This concept is used to reproduce at the model level the modifications performed for an adaptation at the platform level. For example, the code below depicts an adaptation pattern that contains an elementary action modifying the behavior associated with an *addMeeting* port in a diary application.

```
adaptationPattern Synchronization(SynchronizingDiary d1,
                                  SynchronizedDiary d2) {
   modifyPort d1.addMeeting(Meeting m)
     -> if (d2.isFree(m)) then
           d1._call(m); d2.addMeeting(m)
        else
           d1._call(m); d2.printError('Not synchronized')
        endif
}
```

The *Synchronization* adaptation pattern expresses that any addition of a meeting to a synchronizing diary also involves an addition of the meeting to the diary to be synchronized. If this is not possible because the time slot is not free, a synchronization conflict is reported. Note that the *_call* expression refers to a built-in delegation (like *proceed* in AspectJ [9]) that invokes the prior, non-adapted version of *addMeeting*.

Adaptation pattern roles

An adaptation pattern is defined on a set of roles. An **adaptation pattern role** (see the `GenericRole` class in Figure 1) specifies the ports that a component must provide (see the `ProvidedPort` class in Figure 1) or requires (see the `EmittedPort` class in Figure 1) to play this role in an adaptation. For instance, in the previous example, the *Synchronization* adaptation pattern expects two parameters with two different roles: The first parameter must conform to the *SynchronizingDiary* role by providing at least a port that conforms to *addMeeting*. The second parameter must conform to the *SynchronizedDiary* role by providing at least ports that conform to *addMeeting*, *isFree* and *printError*.

2.2 The monitoring criteria: Safety properties

Now that the software entities to monitor are well defined, we need to specify what to monitor. In our case, the monitoring depends on the criteria of safe adaptations. For that, we have identified a set of *safety properties*, which cover a large range of errors: from "assembly inconsistencies" [10], "message not understood" [11] local errors up to more global errors such as "adaptation composition conflicts" [12], "synchronization" and "divergence" [13]. Each safety property is guaranteed by a set of OCL (Object Constraint Language) [14] constraints (operation preconditions) attached to the Satin runtime model.

We suppose that the application initial state is safe (each component and the initial assemblies of these components are safe). If the OCL constraints are checked regarding the adaptation to perform then the adaptation can proceed and we guarantee that the application state remains safe after the adaptation.

Next section explains how to use the Satin runtime model in platforms offering adaptation facilities.

3 Making the model available at runtime

Two approaches can be considered to concretize the model. A first approach consists in adding safety code corresponding to the safety properties in a platform by extension of its model. This is done in two steps: 1) we must map the elements of the Satin model to elements of the target platform, 2) once the target element is identified, we must generate the safety properties's constraints on them [5]. This solution has several drawbacks. First, there is not always a one-to-one correspondence between Satin model element and platform elements. In this case, the generation of the safety properties's constraints is more complicated. Secondly, the minimization of software defects being an important issue in critical application areas, we have used validation and verification techniques to ensure the model correctness with a formal foundation [8]. However, such verifications and validations are lost with this approach since the constraints are regenerated at the platform level: mapping to N platforms implies N revalidations.

Another approach consists in populating the Satin model with platform-dependent application information first and then checking for adaptation safety at the Satin model level. For this, the model is made available at runtime as a service. A service protocol formalizes the way the service can be used and how the service communicates with the platforms. The safety service tells whether an adaptation of an application A is safe or not according to the operations offered by the components and according to the previous adaptations of A, stored as one goes along. This helps to prevent adaptations that would lead the application to an unsafe state. The difficulty with this approach is that we have to manage platform diversity as there is no mapping step. But the gain is that one implementation of the service makes it possible to use the model with several platforms. Moreover, the results of the verification and validation step can be preserved or need to be performed again only once at most.

Section 3.1 describes the service built on top of the Satin model. Section 3.2 explains how we take into account platform diversity.

3.1 Service description

The safety service can be used according to the following process. At each step, a subset of OCL constraints is checked in order to detect if a platform adaptation operation violates one of the safety properties. Figure 2 presents the overall service architecture.

1. Components are registered to the service in order to have a partial representation of the application to monitor. However, the step can be delayed to the first time a component is being adapted (step 3): only the components implied in an adaptation need to be represented in the service state (at the Satin runtime model level).
2. The description of the adaptation to perform at the platform level is registered to the service as an adaptation pattern.
3. An adaptation pattern is applied to a list of components in order to perform at the model level the adaptation triggered at the platform level. The components are registered if not already done.
4. The adaptation pattern of step 3 is unapplied in order for the modifications applied to components to be undone if requested at the platform level. This step is optional.

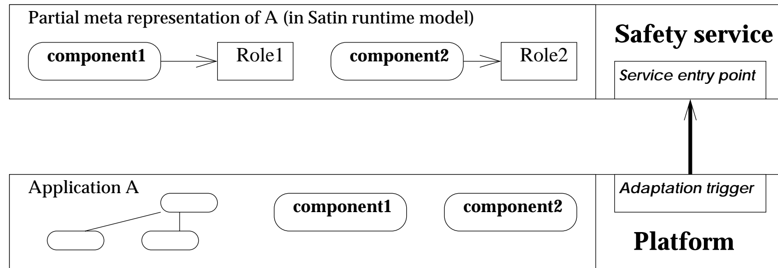

Fig. 2. Safety service architecture : component adaptation and service query

Combining step 3 and step 4 makes it possible to replace components in assemblies. Steps 3 and 4 modify the state of the service to memorize the adaptation of components. These steps are necessary to synchronize the platform and the service so that the history of adaptations is equivalent in the service and in the platform.

The service entry point is described by an IDL Corba interface that allows a platform to use the service without thoroughly knowing its internal mechanisms. Each interface operation is associated with a step of the process: (un)registration of components, creation/destruction of adaptation patterns, (un)application of adaptation patterns, the replacement of components in assemblies.

3.2 Parametrization of service

To evaluate the OCL constraints preserving the safety properties, the service needs information about the monitored application. The way of retrieving and interpreting such information varies from one platform to another one. To take into account platform specificities, the service needs to be parametrized.

Data extraction parametrization point. To populate the model with information about the application, some data need to be extracted. The first time a component is involved in an adaptation at the platform level, a corresponding component is created at the model level (step 1). The initial component roles are deduced from application types using introspection mechanisms which depends on the platform and language.

Type checking parametrization point. Before accepting the application of an adaptation pattern to a set of components (step 3),

each component must conform to the role it has to play in the adaptation. Conformance can be based on different syntactic, behavioral or quality of service criteria [15]. The supported conformance model depends on the platform and language.

Adaptation introspection parametrization point. Applying an adaptation pattern to components (step 3) can be done only if the elementary actions to apply are compatible with the adaptations already applied to those components. The types of elementary actions supported differ from a platform to another. The ways in which the elementary actions are expressed and implemented also vary according to the platforms.

These three parametrization points correspond to the entry points for message exchanges with the platforms and are used by the service to retrieve platform specific information. As for the service entry point, these entry points are formalized by IDL Corba interfaces. Each platform has to provide implementation for these interfaces. Configuring the service for a specific platform consists in specifying which implementation objects the service must use.

A prototype has been implemented in Java. The OCL constraints are not translated into Java code but are interpreted using the Dresden-OCL toolkit [16]. Then a new validation and verification step is not needed provided that the toolkit interpreter preserves the semantics of OCL. Note that even if the prototype is currently used with Java platforms, it can be easily exposed as a web service to interact with non-Java platforms.

4 Discution and future work

The Satin model for adaptation safety monitoring is made available to adaptive platforms as a generic service. Then, the detection of adaptation-related errors is externalized and consists in querying the service to check whether an adaptation request will break the application execution or not. However, the detection of adaptation-related errors can also be done at two other levels. This can be put in place by application developers for each adaptation of their application. This is not a fair solution as the developer is not an expert of the domain. It is also error-prone as each adaptation has to be managed on a case by case basis. It can also be put in place by

the platform that provides the adaptation facilities such as Fractal [17], Sofa [18], JAC [19], Compose* [20] or Noah [21]. This task can then be delegated to an expert of the domain. Though, most of such platforms still lack of formal foundations or force to freeze a part of the application making the hosted applications unavailable to their users. Lastly, these solutions cannot be reused in other platforms.

Self-adaptive systems such as COMPAA [22], PLASMA [23], RAINBOW [24] and SAFRAN [25] monitor and adapt themselves to system errors, changes in the environment or in user preferences. In such systems, adaptations are performed at a model level like in Satin. However, Satin is about introspection not intercession: Satin does not modify the behavior of the monitored application. Model-level adaptation is only done for synchronization purpose between the base level (platform hosting the application) and the model level (Satin safety service). Without the consistency between these two views, the service is not able to compute the adaptation safety anymore. Then Satin should not be seen as a self-adaptive system but as a mean for self-adaptive systems to prevent erroneous adaptations.

Self-adaptive systems detect "changes" and react to changes by "repairing" themselves. In such systems, repairing means adapting the application to the "new context". In our case, repairing only consists in warning the monitored application when an adaptation request is not safe but Satin does not correct the adaptation. In future work, we plan to contribute to decision-making and propose error-free adaptation alternatives when an adaptation request is rejected by the safety service.

References

1. Affer, J.M.: Meta-level programming with coda. In: 9th European Conference on Object-Oriented Programming, Springer-Verlag (1995) 190–214
2. Kickzales, G., deRivières, J., Bobrow, D.G.: The Art of Meta Object Protocol. MIT Press (1991)
3. Muller, P.A., Barais, O.: Control-theory and models at runtime. In: Proceedings of the Models Workshop on Models@Runtime, Nashville, USA (oct 2007)
4. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2) (2006) 25–32
5. Occello, A., Dery-Pinna, A.M.: An adaptation-safe model for component platforms. In: Proceedings of the 3rd International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04), France (2004) 169–174
6. Occello, A., Dery-Pinna, A.M., Riveill, M.: Safety as a service. Journal of Object Technology (2009) to appear in March/April issue.

7. Occello, A., Dery-Pinna, A.M.: Safe runtime adaptations of components: a UML metamodel with OCL constraints. In: First International Workshop on Foundations of Unanticipated Software Evolution, Barcelona, Spagna (2004)
8. Occello, A., Dery-Pinna, A.M., Riveill, M.: Validation and Verification of an UML/OCL Model with USE and B: Case Study and Lessons Learnt. In: Fifth International Workshop on Model Driven Engineering, Verification, and Validation, Lillehammer, Norway, IEEE Digital Library (2008)
9. Kiczales, G., Lamping, J.: Aspectj home page. http://eclipse.org/aspectj (2001)
10. Adámek, J., Plasil, F.: Partial bindings of components - any harm? In: 11th Asia-Pacific Software Engineering Conference, IEEE Computer Society (2004) 632–639
11. Cardelli, L.: Type systems. ACM Computing Surveys (1996) 263–264
12. Hanneman, J., Chitchyan, R., Rashid, A.: Analysis of aspect-oriented software workshop report. Technical report, University of California, Germany (2003)
13. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: 12th ACM Symp. Principles of Programming Languages, New Orleans, LA, USA (1985) 97–107
14. Warmer, J., Kleppe, A.: OCL: The constraint language of the UML. Journal of Object-Oriented Programming (1999)
15. Beugnard, A., J.-M., Plouzeau, N., Watkins, D.: Making components contract aware. In: IEEE Software. (1999) 38–45
16. Wiebicke, R.: Utility support for checking ocl business rules in java programs. Master's thesis, TU-Dresden (December 2000)
17. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Proceedings of WCOP. (2002)
18. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: Proceedings of ICCDS'98, USA (1998)
19. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible and efficient solution for aspect-oriented programming in java. In Yonezawa, A., Matsuoka, S., eds.: Reflection. Volume 2192 of LNCS., Springer-Verlag (2001) 1–24
20. Garcia, C.F.N.: Compose*: A runtime for the .Net platform. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands (2003)
21. Blay-Fornarino, M., Charfi, A., Emsellem, D., Pinna-Dery, A.M., Riveill, M.: Software interaction. Journal of Object Technology **10**(10) (2004)
22. Aniorté, P., Lacouture, J.: Compaa : A self-adaptable component model for open systems. In: 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2008), Belfast, Northern Ireland, IEEE Computer Society (2008) 19–25
23. Layaïda, O., Hagimont, D.: Plasma : A component-based framework for building self-adaptive applications. In: SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications, San Jose, CA, USA (January 2005)
24. Cheng, S.W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. PhD thesis, Carnegie Mellon University (May 2008)
25. David, P.C., Ledoux, T.: An aspect-oriented approach for developing self-adaptive fractal components. In Löwe, W., Südholt, M., eds.: International Workshop on Software Composition (SC). Volume 4089 of LNCS., Vienna, Austria, Springer Verlag (March 2006) 82–97

Runtime Models to Support User-Centric Communication

Yingbo Wang, Peter J. Clarke, Yali Wu, Andrew Allen, and Yi Deng

School of Computing and Information Sciences
Florida International University
11200 SW 8th Street
Miami, FL 33199, USA
{ywang002,clarkep,ywu001,aalle004,deng}@cis.fiu.edu
http://www.cis.fiu.edu/

Abstract. The pervasiveness of complex communication services and the need for end-users to play a greater role in modeling communication services have resulted in the development of the Communication Modeling Language (CML). CML is a domain-specific modeling language that can be used to declaratively specify user-centric communication services. CML models are automatically realized using the Communication Virtual Machine (CVM). The dynamic nature of end-user driven communication results in communication models being updated at runtime. This paper focuses on CML runtime models in the Synthesis Engine (SE), a layer in CVM, which is responsible for synthesizing these models into executable control scripts. We describe how the CML models are maintained at runtime and how they can evolve during the realization of a communication service.

Key words: Communication Model, Model Realization, Model evolution, Runtime

1 Introduction

Electronic communications have become pervasive in recent years. The improvement in network capacity and reliability facilitates the development of communication intensive services and applications. These applications range from IP telephony, instant messaging, video conferencing, to specialized communication applications for telemedicine, disaster management and scientific collaboration [1–3]. Deng et al [4] investigated a new technology for developing and rapidly realizing user-centric communication services to respond to increasing communication needs. We limit the scope of the term communication in this paper to denote the exchange of electronic media of any format (e.g., file, video, voice) between a set of participants (humans or agents) over a network (typically IP). The development process uses a domain-specific modeling language, the *Communication Modeling Language* (CML), which is supported by an automated

model realization platform, the *Communication Virtual Machine* (CVM). The time and cost of developing communication services can be significantly reduced by using the CVM platform for formulating, synthesizing and executing new communication services.

A key part of rapidly realizing communication services is that users can change CML models during execution. In addition, several models associated with the communication service being realized can exist at runtime. These issues raise the question of how to maintain CML runtime models and evolve them in a seamless manner without affecting the current executing communication services. In this paper, we focus on handling CML runtime models in the Synthesis Engine (SE) to address the challenges of model evolution as well as model execution. SE is a layer in CVM, which is responsible for transforming CML models into executable control scripts. These control scripts are executed by the User-Centric Communication Middleware (UCM), a layer below the SE in the CVM. The functionalities of SE includes: (1) maintaining and evolving CML models at runtime, (2) the parsing and interpretation of CML models, and (3) the generation of control scripts.

The rest of the paper is organized as follows. Section 2 introduces the CVM technology. Section 3 presents a motivating scenario form the healthcare domain. Section 4 describes the approach used to manipulate communication models at runtime. Section 5 presents the related work and we conclude in Section 6.

2 Modeling and Realizing Communication Services

In this section we introduce the technology to support the model creation and realization of user-centric communication services.

2.1 Communication Modeling Language (CML)

Clarke et al. [5] developed a language, *Communication Modeling Language (CML)*, for modeling user-centric communication services. There are currently two equivalent variants of CML: the XML-based (X-CML) and the graphical (G-CML). The primitive communication concerns that can be modeled by control CML include: (1) participant, (2) attached device, (3) connection, and (4) data, including simple medium and structured data, which can be transferred. Figure 1(a) shows a simplified version of X-CML using EBNF notation. The EBNF notation represents an attributed grammar where attributes are denoted using an “A” subscript, terminals are bold face and non-terminals are in italics.

A CML model is referred to as a *communication schema* or simply *schema*. A schema consists of two parts: the *control schema* (CS) part which specifies an instance of a topology (participant ids and the types of the exchanged media), and the *data exchange schema* (DS) part which specifies actual media (name or urls) to be exchanged across each connection. We refer the interested reader to [5] for more details.

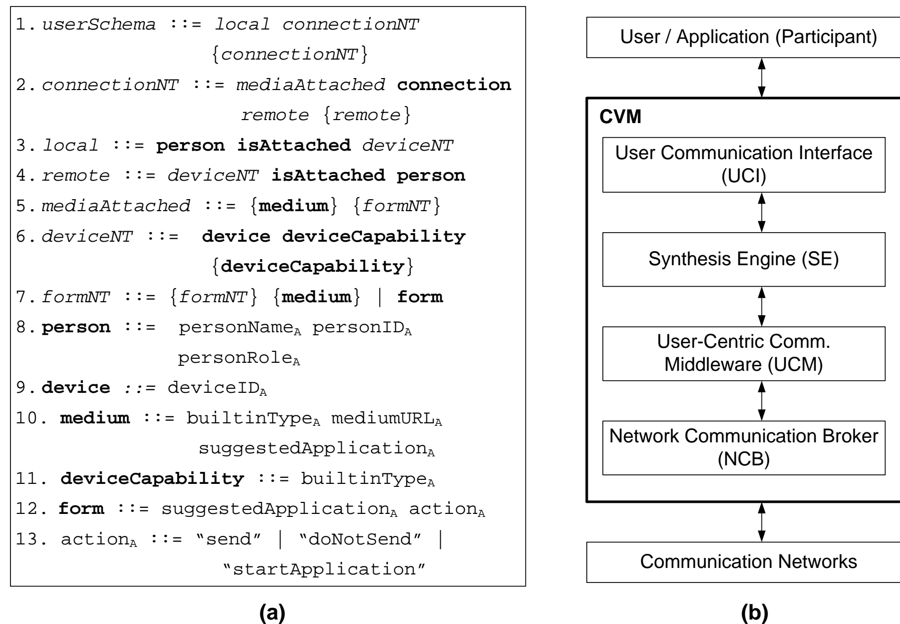

Fig. 1. (a) EBNF representation of X-CML. (b) Layered architecture of CVM

2.2 Communication Virtual Machine (CVM)

The Communication Virtual Machine (CVM) [4] provides an environment that supports the model creation and realization of user-centric communication services. Figure 1(b) shows the layered architecture of the CVM. The CVM architecture divides the major communication tasks into four major levels of abstraction, which correspond to the four key components of CVM: (1) *User Communication Interface (UCI)*, which provides a language environment for users to specify their communication requirements in the form of a schema using X-CML or G-CML; (2) *Synthesis Engine (SE)*, generates an executable script (*communication control script*) from a CML model and negotiates the model with other participants in the communication; (3) *User-centric Communication Middleware (UCM)*, executes the communication control script to manage and coordinate the delivery of communication services to users, independent of the underlying network configuration; (4) *Network Communication Broker (NCB)*, which provides a network-independent API to UCM and works with the underlying network protocols to deliver the communication services.

3 Motivating Scenario

The authors have been collaborating with members of the cardiology division of Miami Children's Hospital (MCH) to study the applications of the CVM tech-

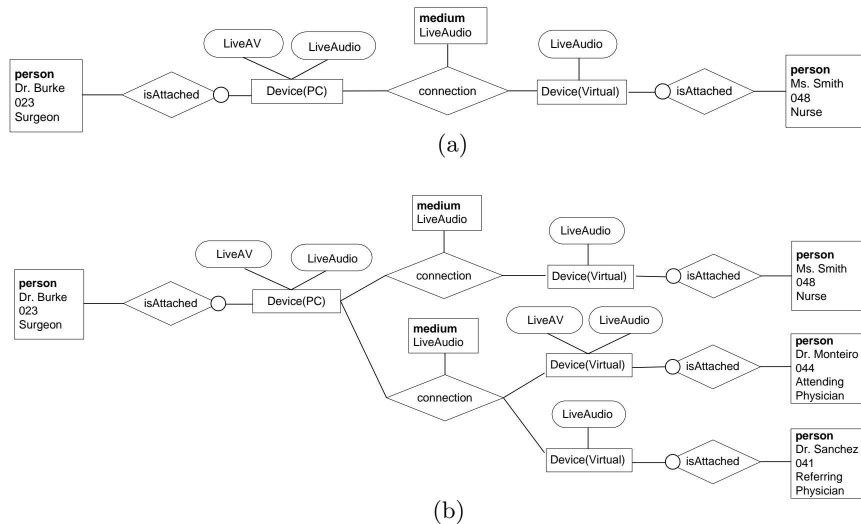

Fig. 2. G-CML models for the scenario. (a) Initial G-CML model of communication between Dr. Burke and Nurse Smith, and (b) G-CML model after all communication connections are established.

nology in the healthcare domain. One of the scenarios we reviewed is described below.

Scenario: After performing surgery on a patient, Dr. Burke (surgeon) returns to his office and establishes an audio communication with Ms. Smith (nurse) to discuss the post-surgery care for the patient. During the conversation with Ms. Smith, Dr. Burke establishes an independent video communication with Dr. Monteiro (attending physician) to obtain critical information for the post-surgery care of the patient. Dr. Burke later decides to invite Dr. Sanchez (referring physician) to join the conference with Dr. Monteiro to discuss aspects of the post-surgery care. Dr. Sanchez's communication device does not have video capabilities resulting in only an audio connection being used in the conference between Dr. Burke, Dr. Monteiro and Dr. Sanchez.

Figure 2 shows two of the three G-CML models created by Dr. Burke during the execution of the scenario. Figure 2(a) shows Dr. Burke's initial request for audio communication with Ms. Smith and Figure 2(b) shows the final G-CML model after Dr. Sanchez is added to the communication. Due to space limitations we do not show the intermediate G-CML model containing only Dr. Burke, Ms. Smith and Dr. Monteiro. A video clip of a similar scenario can be accessed at `http://www.cis.fiu.edu/cml/` that shows an interface for novice users.

4 CML Runtime Models

In this section we provide an overview of how a CML model (schema) is realized by the CVM and the process of synthesizing a schema in the SE. In addition,

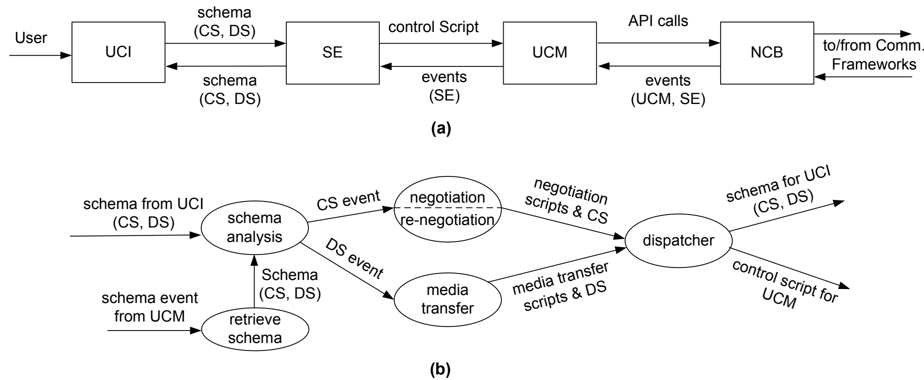

Fig. 3. (a) Execution of a schema in the CVM. (b) Execution of a schema in the synthesis engine (SE). CS - Control schema; DS - Data exchange schema

we describe our approach to handling the different CML models that may exist at runtime.

4.1 Overview of Model Realization

Figure 3(a) shows the process of realizing a CML model (schema) in the CVM. Each participant in the communication has a working CVM. UCI provides the environment for users to create new schemas or load schemas from a repository. SE accepts a schema from UCI or schema events from remote users via the UCM, handles the negotiation process, coordinates the delivery of media, and synthesizes control scripts. UCM is responsible for executing control scripts resulting in API calls to the NCB running on top of a communication frameworks such as Skype [6].

We limit the scope of this paper to the CML models that evolve and are maintained in the SE layer at runtime. Our approach to maintain and evolve the schemas at runtime in the SE involves three main processes: *schema analysis*, *(re)negotiation* and *media transfer* as shown in Figure 3(b). SE accepts a local UCI schema or a UCM event which contains a schema from the remote user, shown on the left side of Figure 3(b). The schema analysis process interprets the schema and generates events based on the runtime control schema (CS) or a data exchange schema (DS). There are two groups of events generated: (1) CS event - passed to (re)negotiation process and (2) DS event - passed to the media transfer process. These two processes work concurrently and both generate control scripts after processing their respective events. The dispatcher sends a control script to the UCM for execution or an updated schema to the UCI to be displayed to the user.

During the execution of a communication schema in the SE there may be several CML control models being manipulated at the same time. These CML models include: (1) the *executing schema* (may have several active connections) which supports the media transfer process to provide a communication service,

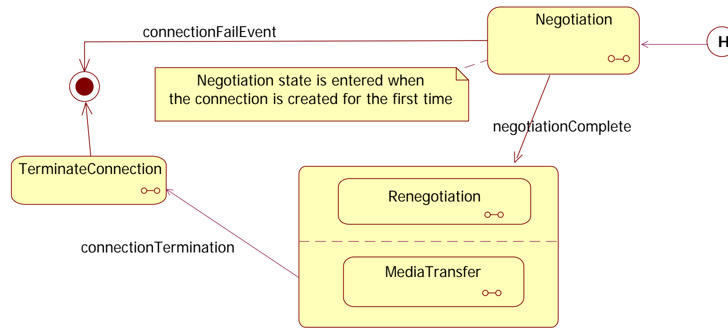

Fig. 4. State machine for a ConnectionProcessor.

(2) an *intended schema* that represents a user's request to change the executing schema (per connection), and (3) a *negotiating schema* providing a transition from the executing schema to the intended schema.

While it is possible to have multiple connections in a CML model, we discuss the runtime CML model in SE based on a single connection because each connection operates independently of each other. Figure 4 shows the high-level state machine for the *ConnectionProcessor* that represents the behavior of a connection. The state machine in Figure 4 consists of four submachines (*Negotiation*, *Renegotiation*, *MediaTransfer* and *Terminate Connection*). The submachines *Negotiation*, *Renegotiation*, and *MediaTransfer* represent the behavior of the processes with similar names in Figure 3.

4.2 Schema Analysis

The schema analysis process compares a received CS/DS for a connection with the locally held CS/DS copy and produces specific events based on the results of the comparison. These events may trigger a transition into the negotiation/renegotiation process, media transfer process, or both (see Figure 3). Figure 5 provides a simplified algorithm of analyzing the CS to illustrate the idea of generating CS events. The algorithm takes the received schema and current schema as input. Based on the source of the received schema (either from UCI or UCM), the role of the local user (whether or not the initiator) and the current schema, it would generate different CS events (line 4, 12, 18, 24 in Figure 5). We have a detailed version of the algorithm that will be presented in a future publication.

4.3 Negotiation/Renegotiation

The *ConnectionProcessor* accesses the *SchemaAnalysis* subprocess to generate CS events and DS events. CS events always affect the negotiation of a new CS or the renegotiation based on an executing CS. DS events carry media transfer request supported by an executing CS. CS events include `initiateNegotiationEvent`, `receivedInvitationEvent`, `sameControlSchemaEvent`, `changeControlSchemaEvent`, and `terminateConnectionEvent`. These events trigger actions for

```
1: analyzeSchema_Control (receivedSchema, currentSchema)
   /*Input: receivedSchema - new schema from the UCI or UCM
            currentSchema - reference to schema in the (Re)Negotiation process
2: if receivedSchema is from UCI then
3:    currentSchema ← receivedSchema
4:    generate initiateNegotiationEvent /*handled by Negotiation/Renegotiation */
5: else if receivedSchema is from UCM and currentUser.isInitiator then
6:    store receivedSchema in an internal recipient list
7:    if all schemas from remote participants are received then
8:       if  mergeSchemas(all schemas) = currentSchema then
9:          generate sameControlSchemaEvent /* the end of negotiation */
10:      else
11:         currentSchema ← mergeSchemas(all schemas)
12:         generate changeControlSchemaEvent /*another round of negotiation */
13:      end if
14:   end if
15: else if receivedSchema is from UCM and !currentUser.isInitiator then
16:   if currentSchema is null then
17:      update currentSchema to include local capabilities
18:      generate receivedInvitationEvent /*display the invitation */
19:   else
20:      if mergeSchemas(receivedSchema, currentSchema) = currentSchema then
21:         generate sameControlSchemaEvent /* the end of negotiation */
22:      else
23:         currentSchema ← mergeSchemas(receivedSchema, currentSchema)
24:         generate changeControlSchemaEvent /*the reply to a negotiation */
25:      end if
26:   end if
27: end if
```

Fig. 5. Algorithm to analyze control schema during negotiation.

creating control scripts and state transitions to handle a non-blocking three-phase commit protocol for schema negotiation [7]. Similarly, DS events trigger the transition of *MediaTransfer* submachine. Using part of the *negotiation* submachine as an example, the `initiateNegotiationEvent` will trigger an action `sendSchemaRequest`, which generates control scripts for sending an invitation, and move the submachine to the *waitingResponse* state. Only the receipt of the `sameControlSchemaEvent`, which indicates all invitees accept the invitation, can trigger the action `sendConfirmation` for creating control scripts to send confirmation of the negotiation and move the *negotiation* submachine from *waitingResponse* to the *negotiationComplete* state.

4.4 Applying Runtime Model to Scenario

Three different intended CML models (schemas) are processed in the motivating scenario (see Section 3). This subsection describes how SE maintains and evolves the executing schema into an intended schema at runtime. Figure 6 shows the SE environment. The intended schema (shown at the top of the figure) is sent by UCI to SE for processing at runtime. This schema reflects that Dr. Burke (`A`) wants to invite Dr. Sanchez (`C`) to join the discussions with Dr. Monteiro (`B`). The executing schema is shown in ellipse labeled *SE Global Schema*, in which Dr. Burke has already established two independent connections, one with Ms. Smith (`D`)(handled by *ConnectionProcessor* C1 shown at the bottom of Figure 6) and the other with Dr. Monteiro (handled by *ConnectionProcessor* C2). Each

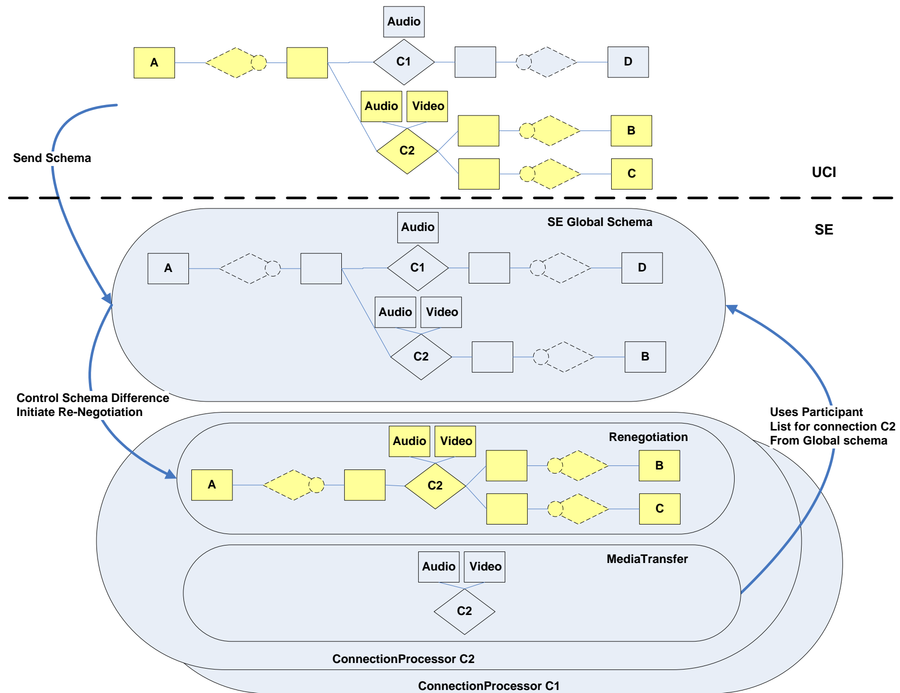

Fig. 6. CML model (control schema) being updated at runtime.

ConnectionProcessor contains two concurrent processes, one for renegotiation and the other for media transfer.

When SE accepts the intended schema from the UCI, it is decomposed into connection schemas and dispatched to the appropriate *ConnectionProcessor*. *ConnectionProcessor* C1 finds no in CS or DS, which means no change in the connection between Dr. Burke and Ms. Smith. For *Connection Processor* C2, it accesses the schema analysis process to compare the new intended CS (UCI schema in Figure 6) with the executing one (the SE Global Schema). The schema analysis process finds differences between these two CSs, which means the currently executing CS needs to be changed. It then generates `initiateNegotiationEvent` to trigger the *Renegotiation* process to initiate schema evolution. The negotiating schema is held in *Renegotiation* process. While the renegotiation is occurring, the *MediaTransfer* process in *ConnectionProcessor* C2 is still supporting the audio and video connection between Drs. Burke and Monteiro using the executing schema. When the renegotiation is complete, a confirmation from Dr. Sanchez's CVM triggers both processes in *ConnectionProcessor* C2. The *Renegotiation* process moves to the *Idle* state and waits for the next request to evolve the CS. The executing schema in ellipse labeled *SE Global Schema* is replaced by the negotiated schema.

4.5 Discussion

Our approach for evolving models is similar to the typical control loop mechanism found in control theory [8]. Schema analysis plays the role of the observer and the *CommunicationProcessor* acts as the controller. Using CML for specifying user-centric communication services, the types of changes that could occur during runtime is predictable and enumerable resulting in a more stable system. Evolution of a currently executing schema into a new schema includes negotiating the schema and switching to the new negotiated schema with minimum effect on the existing services. There are however several remaining questions to be addressed: (1) If the schema evolution is unsuccessful due to an exception, how can the SE rollback to the previous schema? (2) How to keep track of the evolution history of the runtime models? and (3) How to effectively maintain consistency between the executing SE schema and the execution state of the lower layers in the CVM? These open questions would motivate future research in this area.

5 Related Work

Addressing the maintenance and evolution of runtime models in a constantly changing and interactive environment is a major research problem in the area of MDE. Depending on the problem at hand, models might need to evolve to be synchronous with the runtime application through dynamic adaptation, or the runtime system needs to be adapted as the input model evolves. We will see how these problems are addressed in the community.

In Prawee et al [9], the authors developed a framework for co-evolution of system models and runtime applications. As a system is described in the forms of ADLs models and then projected toward an implementation platform, dynamic system adaptation can cause the running system to be out-of-synchronous with its model. The proposed framework enables a system/model evolution and provides architects with consistent views of running systems and their models. We use a different methodology to adapt our CVM at runtime. New communication requirements are represented by an intended CML model and result in a model evolution which leads to the runtime environment adaptation.

Van der Aalst [10] use a generic workflow process model to handle dynamic change of executing processes. Since the change of an executing control flow is a more complicated process, whereby new tasks could be added, old ones replaced, and the order of tasks changed, the number of types of model changes that could occur during runtime becomes significant. How to keep track of different variants of the processes and decide on the safe states for migration is challenging. The paper proposed a generic process model with a minimal representative for each process family to give a handle to deal with these problems. We address similar problems in that we need to manage various CML models during runtime and perform a safe migration of an executing CML model into a new one. However since we are only limited to the communication domain, the types of

possible model changes are fewer and ways of effecting the change could be more dedicated then the general workflow model update.

6 Conclusions and Future Work

In this paper we provided an approach that shows how the Synthesis Engine (SE), a layer in the Communication Virtual Machine (CVM), maintains and evolves runtime models during the realization of user-centric communication services. Three processes were presented that support these activities including: schema analysis, (re)negotiation and media transfer. In addition, a scenario from the healthcare domain was used to show how these processes can be applied during the execution of a communication service. Our future work involves investigating techniques for handling schema rollback, maintaining a schema history and ensuring the consistency of runtime models in the different layers of CVM.

Acknowledgments

This work was supported in part by the National Science Foundation under grant HRD-0317692. The authors thank the reviewers for their insightful comments.

References

1. Burke, R.P., White, J.A.: Internet rounds: A congenital heart surgeon's web log. Seminars in Thoracic and Cardiovascular Surgery **16**(3) (2004) 283–292
2. FEMA: DisasterHelp `http://www.disasterhelp.gov/start.shtm` (May 2008).
3. Cyberbridges: Center for Internet Augmented Research and Assessment `http://www.cyberbridges.net/archive/summary.htm` (Nov2007).
4. Deng, Y., Sadjadi, S.M., Clarke, P.J., Hristidis, V., Rangaswami, R., Wang, Y.: CVM - a communication virtual machine. Journal of Systems and Software (2008) (in press).
5. Clarke, P.J., Hristidis, V., Wang, Y., Prabakar, N., Deng, Y.: A declarative approach for specifying user-centric communication. In: Proceeding of CTS 2006, IEEE (May 2006) 89 – 98
6. Skype Limited: Skype developer zone (Feb. 2007) `https://developer.skype.com/`.
7. Rangaswami, R., Sadjadi, S.M., Prabakar, N., Deng, Y.: Automatic generation of user-centric multimedia communication services. In: Proceedings of IPCCC. (April 2007) 324–331
8. Muller, P.A., Barais, O.: Control-theory and models at runtime. In: Proceeding of 2nd International Workshop on Models@run.time. (Sept 2007)
9. Sriplakich, P., Waignier, G., Meur, A.F.L.: Enabling dynamic co-evolution of models and runtime applications. In: Proceedings of COMPSAC, Los Alamitos, CA, USA, IEEE Computer Society (2008) 1116–1121
10. der Aalst, W.M.P.V.: Generic workflow models: How to handle dynamic change and capture management information? In: COOPIS '99: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems, Washington, DC, USA, IEEE Computer Society (1999) 115

A Framework for Bridging the Gap Between Design and Runtime Debugging of Component-Based Applications*

Guillaume Waignier, Prawee Sriplakich, Anne-Françoise Le Meur, Laurence Duchien

Université Lille 1 - LIFL CNRS UMR 8022 - INRIA
59650 Villeneuve d'Ascq, France
{Guillaume.Waignier,Prawee.Sriplakich,
Anne-Francoise.Le_Meur,Laurence.Duchien}@inria.fr

Abstract. One concern when building application by assembling software components is to validate component interactions, *e.g.*, to ensure that components exchange compatible messages. This validation requires examining data values that are only known at runtime. In current practice, this validation is often performed manually at the code level, *i.e.*, architects need to insert validation code into the application code. This situation makes the interaction validation costly. Moreover, few platforms provide sufficient tools for supporting this validation.
As a solution, we propose CALICO, a model-based framework for runtime interaction validation. CALICO enables architects to specify validation concerns in the application model. It automatically propagates this specification to application code so that component interactions in the application can be checked at runtime. Based on the detected errors, CALICO allows architects to revisit the design to fix the detected errors, and then to repeat the runtime validation in an iterative process. This paper focuses on the integration of tools in CALICO for linking between validation specification at design time and validation realization at runtime. Moreover, we show how to extend CALICO to support multiple platforms with small development effort.

1 Introduction

CBSE is a widely used paradigm for creating complex software. It consists in building an application by assembling software component [1]. To ensure application reliability, CBSE software development usually includes checking the consistency of *component interactions*, which concerns the values of messages exchanged by components, and the order in which components exchange them. This validation aims to detect errors, such as an incompatibility of message values exchanged by components.

In a typical software process, architects perform this validation while iterating over the design, implementation and debugging tasks. The design task consists in elaborating application models describing a component assembly, which we refer to as *Architecture Description (AD) models*. This task also includes static validation of the AD models to ensure the consistency of component interactions [2]. The implementation task corresponds to generating and writing code for each component. The debugging task aims

* This work was partially funded by the French ANR TL FAROS project.

at checking whether the modeled application can execute correctly on a given platform. It consists in loading the components specified in the AD models on the target platform, and in detecting *runtime errors* on component interactions. This detection requires analysis of data whose values can only be known at runtime. To take into account runtime errors detected during the debugging task, architects reiterate over the design, implementation and debugging tasks, *i.e.*, they edit the AD models to fix the problems, reimplement the added or modified components and redebug the application.

These three tasks require using the combination of several software artifacts, such as models, code, configuration files, and involve multiple tools, such as model editor and model analysis tools for the design task, a code generator tool for the implementation task and a deployment tool for the debugging task. However, provided by different vendors, these tools are usually poorly integrated, making difficult to architects to use them in an iterative way. Indeed, the architects usually have to manually transfer data among tools and to convert data among the formats required by the different tools. This task is tedious and prone to errors.

Furthermore, few platforms provide a sufficient tool set for enabling architects to develop a reliable application. For example, platforms such as CCM [3] and Fractal [4] provide no support to statically check component interaction inconsistency at design-time. Thus, when developing an application on such a platform, architects suffer from the lack of tools for detecting inconsistencies in an early phase of the software development process. Moreover, few platforms provide mechanisms for debugging component interactions at runtime; consequently, architects can suffer from workload in coding the debugging support in their application code.

To enable architects to perform efficient iterative software development on a platform of their choice, we propose CALICO, the Component AssembLy Interaction Control framewOrk. CALICO provides architects with a set model-based tools for editing AD models and checking model consistency during the design task, for generating skeleton code during the implementation task and for validating component interactions at runtime during the debugging task. First, CALICO enables architects to address *validation concerns* at the model level, *i.e.*, it allows architects to specify, on the AD models, the component interactions they need to check. For example, architects can specify the assertion that the data a component receives must be of valid format. CALICO ensures that this verification is performed at runtime, *i.e.*, during the debugging task. When an error is detected, architects can use CALICO to fix the problems in the AD models. Each modification in the models is propagated to the running system by performing dynamic updates of the component assembly. Architects can reiterate over the design, implementation and debugging tasks until the system becomes stable.

This paper focuses on the set of tools that CALICO proposes for bridging the gap between the design and the runtime debugging. When designing CALICO tools, we aimed to provide genericity, *i.e.*, the tools manipulate AD concepts that are common to several platforms, and extensibility, *i.e.*, the tools can be extended with functionalities to validate applications running on different platforms. We explain how these tools are integrated to enable architects to perform iterative software process in a continuous way. We describe how CALICO can be extended to support multiple platforms, so that architects can gain the ability to develop reliable software even on platforms that cur-

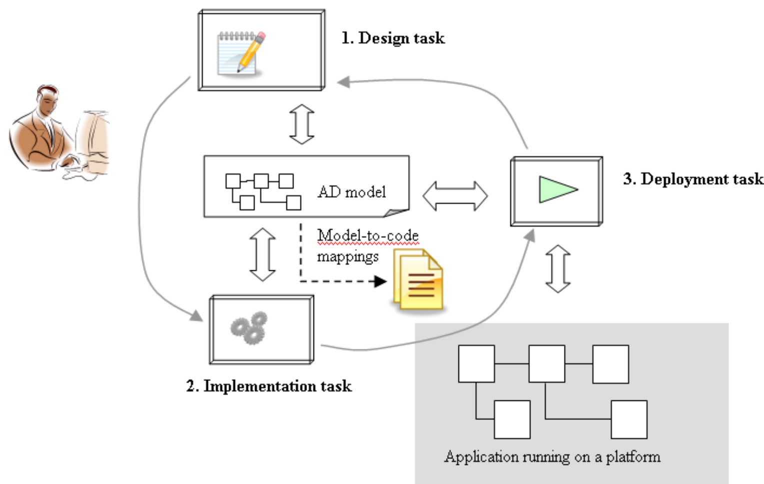

Fig. 1. Overview of CALICO

rently lack support for component interaction validation. Complementary to this paper, details on the AD model semantics and the model analysis techniques for validating component interactions are presented in our previous papers [5] and [6].

This paper is organized as follows. Sections 2 gives a rapid overview of CALICO and illustrates how it can be used in an iterative software development process. Sections 3 describes how CALICO handles the multi-platform support and the iterative software development process. Finally, Section 4 compares CALICO with other work and Section 5 concludes.

2 Iterative Software Development

CALICO is a framework for designing and debugging component-based applications on multiple platforms. It covers three tasks in an iterative software development process: design, implementation and debugging (c.f. Figure 1). CALICO integrates these tasks to enable continuity between them. This integration is based on using common application's AD models. The AD models are created during the design task, used to both generate code during the implementation task and finally load the application during the debugging task.

2.1 Overview

In the **design task**, CALICO enables the architect to design and analyze the specification of an application. The application is described with a set of AD models containing both the Platform Independent Model (PIM) part and Platform Specific Model (PSM) part. The PIM part includes the *system structure model*, which depicts the application architecture in terms of components and connections. It also contains three models to

express, respectively, the structural constraints, the control flow going in and out of components, and the constraints on the values of the exchanged messages. The PSM part refines the PIM part by adding platform-specific details.

During the design phase, CALICO offers a platform-independent approach to address validation concerns. First, to address static validation, it provides architects with a tool to statically verify model consistency regarding component interactions. Second, to address runtime validation, CALICO offers also the notion of *validation points*, which specify the interactions that needs to be checked at runtime, *i.e.*, the value and order of the messages exchanged between connected components.

In the **implementation task**, CALICO provides the generation of skeleton code, *i.e.*, component interfaces and component implementation classes, which are to be filled with business code. This code generation reduces manual errors that architects could have made in code writing. It guarantees that the component implementation respects the component specification expressed in the system structure AD model.

CALICO also addresses validation concerns by translating validation points into components, called *interceptors*. The interceptors are responsible for performing, at runtime, the checks described in the validation points. Furthermore, CALICO automatically instruments the whole application code to add the debugging support. The instrumented code reifies runtime information, *e.g.*, control flow of component interactions, that the interceptors require for performing the checks. This instrumentation is generic enough to allow the architect to develop a reliable system on any given platform, even if the platform lacks support for reifying the runtime information. From the architect's viewpoint, the instrumentation is driven by models, and the mechanisms to realized the instrumentation are transparent to the architect.

During the **debugging task**, CALICO loads the application onto the underlying platform and instantiates the needed interceptors. At runtime, the interceptors check the component interactions and report to the architects the detected errors. The error detection may lead architects to revisit the design task, *i.e.*, updating the AD models, for fixing the detected error. Then, when the architect iterates over the implementation and debugging tasks, CALICO dynamically propagates the changes into the running system, without reloading the whole system.

2.2 An Iterative Scenario: the PHR System

To illustrate the use of CALICO, we present an example software development scenario for the French Personal Health Record system (PHR) system [7]. The PHR system aims to enable heterogeneous clients, used by health-care staffs, to access Health Record documents stored in servers. Several clients are connected to the Web Portal service, which is in charge of dispatching the client requests to the corresponding Data Store servers. In this system, each client only supports particular document formats, *e.g.*, HTML and plain text, while the format of documents provided by heterogeneous servers may be different. Consequently, when an architect executes tests on the system, he/she notices the following errors: some clients receive documents in unsupported formats.

Consider now that the PHR system has been designed and implemented with CALICO (cf. Figure 2). To debug the PHR system, the architect needs to identify which clients, servers and documents are causing the errors. Accordingly, the architect can

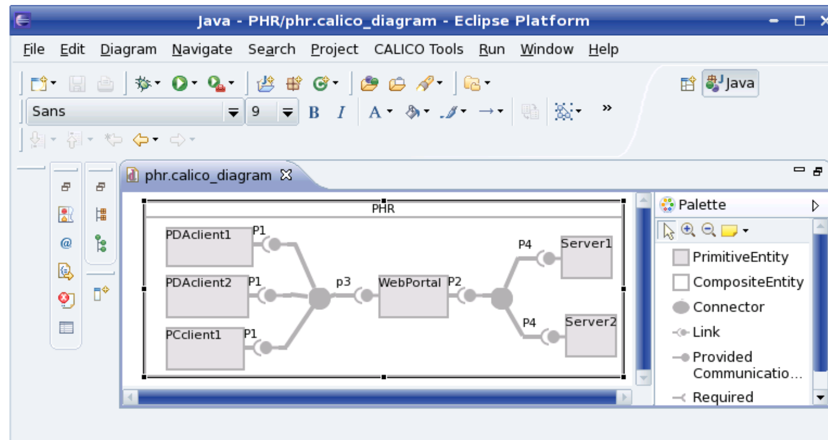

Fig. 2. PHR system

place validation points on the server ports, in order to detect errors as early as possible. The validation points express the checks to perform on the documents returned by the servers, to identify potential incompatibility with requesting clients. These checks rely on control flow information to determine the client involved in each control flow. CALICO propagates the validation points into the running system by generating and loading the interceptors. After the interceptor insertion, the architect can run tests again. The inserted interceptors enable the architect to obtain the source of the errors.

Having identified the error source, the architect is able to provide a solution to fix the problem. This solution consists in inserting Data Converter components between the Web Portal and incompatible clients to convert the data returned by the servers into a format compatible to the client, *e.g.*, converting Microsoft Word into plain text documents. This solution is realized during another software development iteration, *i.e.*, using CALICO to update the design and to propagate the changes to the running system.

3 The CALICO Architecture

We describe in this section two parts of CALICO: the model and tool parts. With respect to the model part, we present the system structure AD metamodel that is used by architects to define an application architecture, *i.e.*, AD model. Based on this metamodel, we define the Update metamodel representing the changes in the AD model performed by the architects at each software development iteration. The tool part concerns the framework architecture that allows the integration of tools involved in the design, implementation and debugging tasks, and enables the framework extension to support multiple platforms.

3.1 The Metamodels Part

The system structure AD metamodel offers metaclasses that represent the concepts of Component, Port and Connector, which are common concepts in component-based plat-

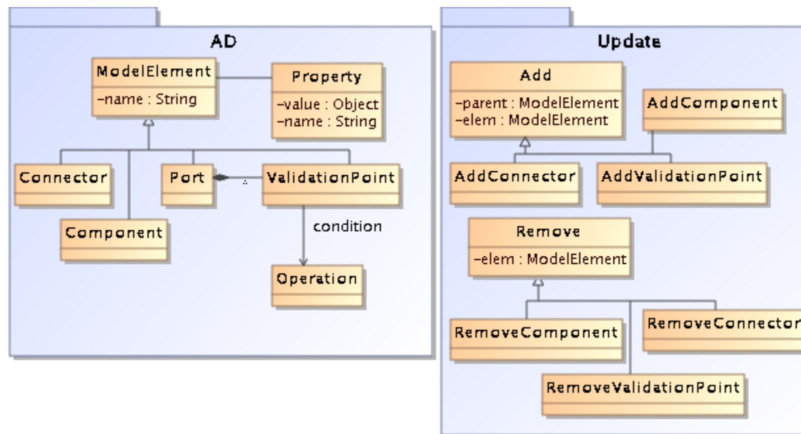

Fig. 3. AD metamodel and Update metamodel

forms (cf. Figure 3). These concepts have already been presented in [6], in this paper we focus on the concepts concerning the iterative software development support and the multi-platform support.

Linking the design to runtime debugging. The system structure AD metamodel enables architects to insert validation points on component ports, which specifies that a message sent or received through the port needs to be reified for debugging purpose. A validation point contains a message filtering condition. This condition is a boolean operation that is evaluated on the data values contained in the messages. In the PHR example, this operation checks if the documents sent by the server is compatible with the client. The condition specification greatly reduces the architect's workload when debugging, by selecting the reified information that the architect needs to analyze.

Supporting Multiple Platforms. Each platform usually requires extra information in the AD model in order to define platform-specific configuration, such as component implementation or connection type, *e.g.*, synchronous, asynchronous, etc. [8]. CALICO offers means to specify Platform Specific Model (PSM) properties and to validate them. The AD metamodels adopts the concept of *property* of Acme [9] for defining these PSM properties. A property is a name-value information entity that can be attached to a model element. The validation is encapsulated in a *Platform Profile*. A Platform Profile defines a set of PSM properties that an architect is allowed to specify, and the set of constraints, written in OCL, that must be satisfied by the application model in order to make sure that the application can be loaded on a given platform.

For example, the Platform Profile of CCM defines the property `impl-class` for specifying the implementation class of each component, and the property `type` for specifying whether the component port is a `facet`, `receptacle`, `event_sink` or `event_source`. This Platform Profile specifies also that only ports with compatible types can be connected.

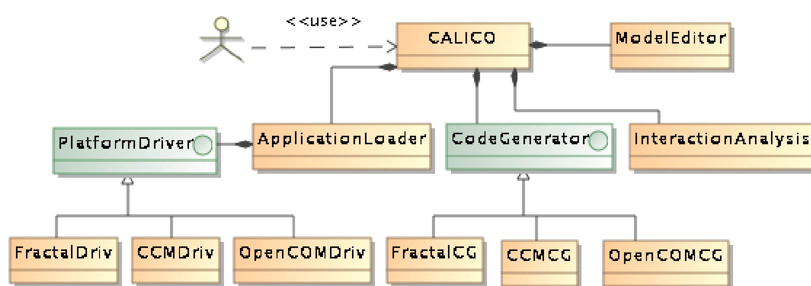

Fig. 4. Tool integration in CALICO

Supporting Iteration. Based on the AD metamodel, we define the Update metamodel, which represents the changes between two versions of an AD model (cf. Figure 3). The Update model contains the sequence of `Add`/`Remove` operations that define the addition/removal of model elements, *i.e.*, components, connectors and validation points, to/from the AD model. The `Add` operation specifies the model element to be added and the location to add it, *i.e.*, the parent model element. For example, the operation `AddValidationPoint` specifies the validation point to be added, and the component port where to attach the validation point.

The Update model is only used internally within CALICO, the architect does not need to have access or edit the instance of this metamodel. Rather this instance is generated automatically by comparing the two model versions: the version representing the running application and the version that has been modified by the architect.

3.2 Tool Part

CALICO offers a set of tools for supporting the design, implementation and debugging tasks (cf. Figure 4). During the design task, an architect uses the *Model Editor* tool to edit the AD model, and refine it with PSM properties. Then, the *Interaction Analysis* tool checks the constraints on the AD model to ensure consistency. During the implementation task, the architect uses the *Code Generator* tool both to generate the skeleton code of the components and to instrument the application code to support debugging. During the debugging task, the *Application Loader* tool updates the running application, accordingly to the AD model.

Linking the design to runtime debugging. To support debugging, CALICO offers translation of validation points into interceptors in the running application. An interceptor is a non-business component inserted between two ports of interacting business components. It provides a pair of client and server ports to forward the messages sent and received by the business components. Its role consists in evaluating if the method arguments of the port respects the condition specified by the validation point. We have chosen to realize the interception mechanisms using regular components for genericity purpose, *i.e.*, this approach can be realized on any platforms. Moreover, this approach enables us to dynamically add interception mechanisms to the running application.

Supporting Multiple Platforms. We propose the notion of plugins to integrate platform-specific functionalities into the framework. A plugin contains the following elements: a Platform Profile, a Code Generator and a Platform Driver.

The *Platform Profile* is an OCL file defining the PSM properties that are checked by the Interaction Analysis tool during the design task.

The *Code Generator* implements the Code Generator interface, defined in CALICO. It defines three main operations. The operation `genSkeleton` generates the skeleton code of the business components from the system structure AD model. The operation `genInterceptor` produces the implementation of the interceptors. Its implementation can be based on platform-specific code templates. The operation `instrument` is used for instrumenting the whole application code for inserting the debugging support, *e.g.*, reifying control flow. Its implementation is based on aspect weaving.

The *Platform Driver* implements the Platform Driver interface. Its role consists in masking the heterogeneity of platform APIs from Application Loader. It defines seven operations for initializing the platform, creating/removing components/connectors and pausing/resuming components. The Application Loader tool masks the notion of validation points from the Platform Driver by translating them into interceptor components. The Platform Driver does not need to make a distinction between the interceptor components and the business components. Thus, the developer of the driver only needs to provide mechanisms for realizing the `Add`/`Remove` operations of components/connectors. This approach enables the validation point translation mechanism to be refactored, so that it can be reused in multiple platforms.

In the design of the Platform Driver interface, we address the problem that platform APIs require heterogeneous parameters. As a solution, we make the AD model, which contains PSM properties, available to the Platform Driver. This approach enables architects to extend the AD model so that it contains sufficient platform-specific information for being loaded on an underlying platform. For example, in order to instantiate a component, the driver can access the PSM properties of the component definition for identifying the binary code of the component.

Supporting Iteration. To support iteration, CALICO enables an architect to update the running application through the system structure AD model. The Application Loader offers an operation for updating the running application, accordingly to the new AD model that the architect has edited. This update is performed as follows. First, the Application Loader generates the Update model by comparing the new AD model with the one it has preserved since last reconfiguration, *i.e.*, the AD model of the currently running application. The comparison consists in matching structural elements of both models. The elements that do not match are identified as added or removed elements. This comparison mechanism is applicable for both hierarchical-component models, *e.g.*, Fractal [4], and flat-component models, *e.g.*, OpenCCM [3], OpenCOM [10]. Furthermore, if the behavior of a component changes, this component is identified as removed and added, since in our approach we consider components as black boxes. Accordingly, the only way to change the component behavior is to change the component code and to reload the component in the system. In the second step, the Application Loader translates the operations "add/remove validation points" of the Update model into operations for adding/removing interceptor components. Finally, the Plat-

form Driver executes the operations in the Update model. To avoid that an update to a model puts the system into an inconsistent state, the Update model is first applied on a clone of the AD models, which is statically verified, before applying the update on the running system.

4 Related Work

Several script languages, such as FScript [11], are intended to support dynamic reconfiguration of component-based applications. In these approaches, architects define system update in terms of operations such as adding/ removing components. These approaches do not prevent architects from writing a script that creates system inconsistencies. In CALICO, the Update model, equivalent to a script, is automatically generated by comparing the model of the running system and the new model designed by architects. Our approach is result-oriented: it allows architects to check the preview of the update result. Furthermore it simplifies the architect's task in updating the application design, by eliminating the architect's need to learn the script language.

Plastik is an ADL/Component runtime integration meta-framework [12]. Like CALICO, it offers a platform-independent language to describe the software architecture, by using an extension of Acme/Armani [9], and mechanisms for loading the application on the underlying platform. However, to our knowledge, it is implemented only for the OpenCOM platform [10]. Moreover, there is a gap between the design, implementation and debugging tasks. First, architects must manually write the component code accordingly to the architecture description. Second, architects have to implement their components so that they are able to reify runtime information. These manual efforts can be error-prone. On the contrary, CALICO automates the transition between the design, development and deployment tasks, by automatically generating skeleton code and interceptor code.

5 Conclusion

This paper presents CALICO a generic and extensible framework for bridging the gap between the design, implementation and debugging tasks. Building such a framework requires dealing with different information involved in each of the tasks. The model task requires high-level specifications of component structures and behaviors, for enabling consistency checking; the implementation and debugging tasks require platform-specific, low-level specifications, which enable the executability of the modeled application. To tackle this challenge, we propose a tool integration approach based on generic, yet extensible, AD models. This approach reduces the workload of architects in realizing transitions between the design, implementation and debugging tasks: from the architect point of view, it may look like that the execution and debugging tasks are directly performed on the application model. Moreover, the multiple platform support of CALICO enables an analysis tool to be written once and for all, and each platform supported by CALICO can then benefit from this tool.

CALICO has been implemented and is fully integrated with Eclipse [1]. This allows architects to do all iterative development tasks without leaving the integrated environ-

[1] CALICO is available at http://calico.gforge.inria.fr

ment, *i.e.*, graphically designing the application model, checking model consistency, examining and correcting the model inconsistency, generating skeleton code, adding business code, compiling the code, and executing and debugging the application. The entire framework uses EMF to manipulate the models, which are the core data used by all tools.

We have successfully extended the framework with plugins for three platforms: OpenCCM [3], Fractal [4] and OpenCOM [10]. For each plugin, the implementation efforts consist in studying the platform-specific component model to define the Platform Profile, in developing the code templates and code instrumentation aspects to implement Code Generator, and, in implementing the Platform Driver, based on the Platform's API. As experience feedback, we have found out that parts of the code templates and aspects can be reused in several platforms. Moreover, by refactoring the Application Loader's mechanism that translates validation points to interceptors, we were able to reuse it in several platforms. Our experience shows that, extending a CALICO to support new platforms, OpenCCM or OpenCOM, can be done with small effort, *i.e.*, one man-week for each platform.

In near future, we plan to add the support for iterative development on service-oriented platforms, in particular those based on Web Services, such as SCA. We also consider implementing Platform Drivers that support complex component connectors, such as stream-based, secured and broadcasting connectors.

References

1. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. on Software Engineering **26**(1) (January 2000)
2. Hoare, C.: Communicating Sequential Processes. Prentice Hall (June 2004)
3. OMG: CORBA Component Model, v4.0, formal/06-04-01. (April 2006)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in Java. In: Proceedings of the 7th International Symposium Component-Based Software Engineering. Volume 3054 of LNCS., Springer (May 2004)
5. Waignier, G., Le Meur, A.F., Duchien, L.: Architectural specification and static analyses of contractual application properties. In: Proceedings of the 4th International Conference on the Quality of Software-Architectures (QoSA 2008). (2008) To appear.
6. Waignier, G., Sriplakich, P., Le Meur, A.F., Duchien, L.: A model-based framework for statically and dynamically checking component interactions. In: Proceedings of the ACM/IEEE 11th International Conference on MODELS 2008. (2008)
7. Nunziati, S.: Personal health record www.d-m-p.org/docs/EnglishVersionDMP.pdf.
8. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: ICSE. (2000) 178–187
9. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Foundations of Component-Based Systems. Cambridge University Press (2000)
10. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: Opencom v2: A component model for building systems software. In: IASTED Software Engineering and Applications. (2004)
11. David, P.C., Ledoux, T.: An aspect-oriented approach for developing self-adaptive fractal components. In: Software Composition. (2006) 82–97
12. Batista1, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Proceedings of the 2nd European Workshop (EWSA 2005). (2005)

A Model-Driven Approach for Developing Self-Adaptive Pervasive Systems*

Carlos Cetina, Pau Giner, Joan Fons and Vicente Pelechano

Research Center on Software Production Methods
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
{ccetina,pginer,jjfons,pele}@pros.upv.es

Abstract. Adaptive systems are generally difficult to implement, and their quality depends much on the designer experience or creativity. This paper presents a model driven approach to develop adaptive systems by means of run-time models. Our approach applies techniques from the Software Product Lines (SPLs) to address the different requirements of evolution and involution scenarios in Pervasive Systems. Finally, we show how models drive the adaptation in order to dynamically change the system architecture.

1 Introduction

Pervasive computing is defined as a technology that "weaves itself into the fabric of everyday life until it is indistinguishable from it" [1]. To be successful, the pervasive computing functioning should be transparent to the user. Such transparency is achievable only if the software frees users from having to repair and reconfigure the system when faults or changes occur in the environment.

Pervasive systems are highly dynamic and fault-prone since their components are liable to appear and disappear at any time. On the one hand, new kinds of entities (devices such as media players, light sensors or fire alarms) can be incorporated to the system. When a new resource is added to the system, the pervasive system should adapt itself to take advantage of the new capabilities introduced by this resource. On the other hand, existing entities may fail or be disconnected from the system for a variety of reasons: hardware faults, OS errors, software bugs, network faults, etc. When some resource is removed, the system should adapt itself in order to offer its services in an alternative way to reduce the impact of the resource loss.

In a previous work [2], a methodology based on SPLs principles was defined to cope with adaptivity of Pervasive Systems. This approach is based on the reuse of the knowledge from the design of SPLs to support adaptivity in the resulting systems. By means of model transformations, the SPL knowledge is systematically reused at run-time.

* This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and cofinanced by FEDER.

The present work is focused on providing a model-based support for some adaptation scenarios very common in Pervasive Systems (evolution and involution scenarios). These scenarios have different requirements regarding adaptation, and the way in which models are handled at run-time should consider those particular requirements.

The contribution of this work is twofold. On the one hand, a model-based approach is introduced in order to organize the knowledge required for adaptation according to the specific demands (support for involution and evolution scenarios) of pervasive systems. In this way, adaptation knowledge is separated from the structure of the system and different adaptation mechanisms can be offered depending on how much critical the adaptation is. On the other hand, we present how models drive the system adaptation within the context of each scenario.

The remainder of the paper is structured as follows. Section 2 gives an overview of the different models used to structure the knowledge about the system. Section 3 defines the architecture proposed for the kind of system this work is dealing with. Section 4 defines for different scenarios how to achieve the adaptation of a system that follows the introduced architecture using the presented models. Section 5 discusses related work. Finally, Section 6 presents some conclusions to the paper.

2 The Models for System Adaptation

The present work considers system adaptation as a reaction to a change in system resources. Therefore, two different kind of scenarios are considered: *evolution scenarios* (a resource is added) and *involution scenarios* (a resource is removed).

Evolution and involution scenarios have different requirements regarding adaptation. On the one hand, in involution scenarios time becomes critical since these scenarios are normally related to failure-recovery. For example, if an alarm system fails in a smart home, an alternative system (e.g., house lights blinking) should be used immediately as a backup. On the other hand, evolution scenarios are not so demanding in this aspect, and even the opinion of the final users can be considered (interacting with the users or considering their preferences) to provide a better adaptation according to user needs.

In order to enable system adaptation, it is required a knowledge about (1) the current state of the system and (2) the possible ways of changing it. In the present work, models are used for both purposes, being queried at run-time to perform the adaptation. On the one hand, models are used for capturing the state of system components and the communication channels among them. On the other hand, the space of possible system changes is captured by means of feature modeling techniques.

The consequences of a change in the system (e.g., enabling the security feature) can be obtained by reasoning over a model that captures all the possible system features and their dependencies. This is acceptable for evolution scenarios where the system is being upgraded. However, since involution scenarios

require an immediate adaptation, information required for the system reaction is precalculated in models. In this way, the effort of reasoning over the different possibilities of adaptation is avoided. More detail about the different models involved in the proposal is provided below.

- **Feature model.** Feature Modeling is a technique to specify the variants of a system in terms of features [3]. In feature models, features are hierarchically linked in a tree-like structure and are optionally connected by cross-tree constraints. This model describes the possible system functionality and its dependencies in a coarse-grained manner. The impact of activating a feature is captured in this model. For example, if a security system is installed, other features depending on a security system such as presence simulation can be activated.
- **Realization model.** Realization model defines relationships between features and components of the architecture. Atlas Model Weaving has been extended [4] to define the *default* and *alternative* relationships. In this way, some components of the architecture are labeled as default or alternative components for supporting a certain feature. In case of failure of a component, this model permits to quickly find an alternative to replace it.
- **Component model.** This model represents the different components that conform the system. Component state is captured in the model. This model is in synchronization with the system since components make use of this model to store and retrieve their state.
- **Structural model.** This model represents the communication channels established between the different components of the system. Since this work deals with highly dynamic systems, the connection among components change quite often. This model reflects both, the possible connections and the ones that are in use in the current state of the system.

The introduced models have been structured in this way in order to decouple system adaptation (Feature and Realization models) from system description (Component and Structure models). In addition, precalculated information to better support involution scenarios is isolated in a model (Realization model). Next sections describe the use of these models to achieve system adaptation.

3 The Model-Based Adaptation Approach

To perform adaptation, our approach is based upon a framework for adaptive systems proposed in [5] by analyzing common terminology and synergy between different approaches. This framework introduces the roles of (1) triggers which specify the event or condition that causes the need of adaptation; (2) adaptation actions which realize the actual adaptation; and (3) adaptation rules that define which triggers cause which adaptation actions. In our approach, these rules are driven by run-time models to modify the system architecture using adaptation actions.

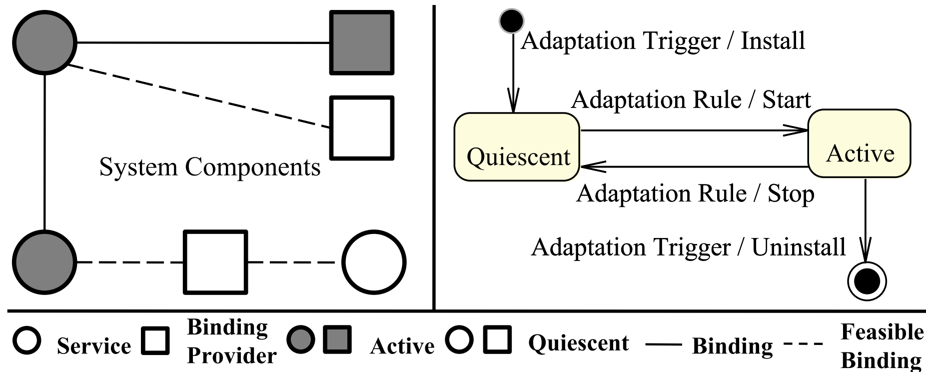

Fig. 1. The adaptation architecture.

3.1 The System Architecture

In order to allow a flexible adaptation process, we have considered an architecture based on communication channels (called bindings). This architecture for the final system allows an easy reconfiguration process since communication channels can be established dynamically between the components that form the system (see left of Fig. 1). These components are classified in Service and Binding Providers as follows:

- **Service.** A *Service* coordinates the interaction between resources to accomplish specific tasks (these resources can be hardware or software systems);
- **Binding Provider.** A *Binding provider* (BP) is a resource adapter that handles the issues of dealing with heterogeneous technologies. The BP provides a level of indirection between Services and resources. Resource operations interact with the environment (sensors and actuators) and provide functionality from external software systems. Services coordinate these resource operations to offer high-level functionality. If the resource operations do not match the Service expectations, then a BP is used to adapt these operations. Hence, the BPs decouple Services from resource operations.

For example, in a smart home a security service is composed of several resources such as presence sensors, movement detectors, sirens, contact detectors, SMS senders, silent alarms and so on. The security service coordinates the behaviour of all these resources.

3.2 Adaptation Actions

The system architecture has to be modified as a result of the dynamic adaptation. Old components must be dynamically replaced by new components while the system is executing. The adaptation actions are in charge of this dynamic reconfiguration. These actions deal directly with the system components by means of the following operations: Component State-Shift and Component Binding.

1. **Component State-Shift** Kramer and Magee [6, 7] described how in an adaptive system, a component needs to transit from an active (operational state) to a quiescent (idle) state in order to perform the system adaptation. We have applied this approach to our systems by means of the OSGI framework [8]. The OSGI Framework defines a component life cycle where components can be dynamically installed, started, stopped, and uninstalled (see right of Fig. 1). On the one hand, *Triggers* are in charge of perform the install/uninstall operations. For example, when a resource fails or a new resource is installed in the system. On the other hand, *Adaptation Rules* are in charge of perform the start/stop operations. For example, when a Binding Provider must be activated to handle a new resource.
2. **Component Binding** Once a component transits to an active state, it needs bindings with other components. These bindings are implemented by using the OSGI Wire Class (an OSGI Wire is an implementation of the publish-subscribe pattern oriented to dynamic systems). The OSGI Wires establish communication channels between components to send messages one another.

Adaptation actions provide the basics operations to dynamically change the system architecture. Adaptation rules orchestrate the execution of these actions by means of the run-time models. The next section details how the adaptations rules queries the models in order to apply the adaptation actions.

4 Adaptation Rules

In a nutshell, an adaptation rule is in charge of (1) handling the adaptation triggers, (2) gathering the necessary knowledge from the run-time models and (3) applying the adaptation actions.

As we state above, evolution and involution scenarios present different requirements. In involution scenarios the system must provide an autonomic response in a reduced amount of time. While in evolution scenarios, the system does not present the same time requirement and even the user might assist the adaptation. To fulfil theses requirements, we have defined two kinds of adaptation rules taking into account the type of scenario.

4.1 Adaptation in Evolution Scenario

When a component is plugged-in, first the adaptation rule queries the feature model for which new features could potentially be activated. Then the user confirms the features activation. Furthermore, activating new features can fulfil other feature constraints which might be enabled. Therefore, each time the user confirms a feature activation, the adaptation rule queries the feature model for new features. Finally, the Component and Structure Models drive the adaptation actions in order to dynamically reconfigure the system architecture and support the new features. The steps to perform this adaptation (see Fig. 2) are detailed next:

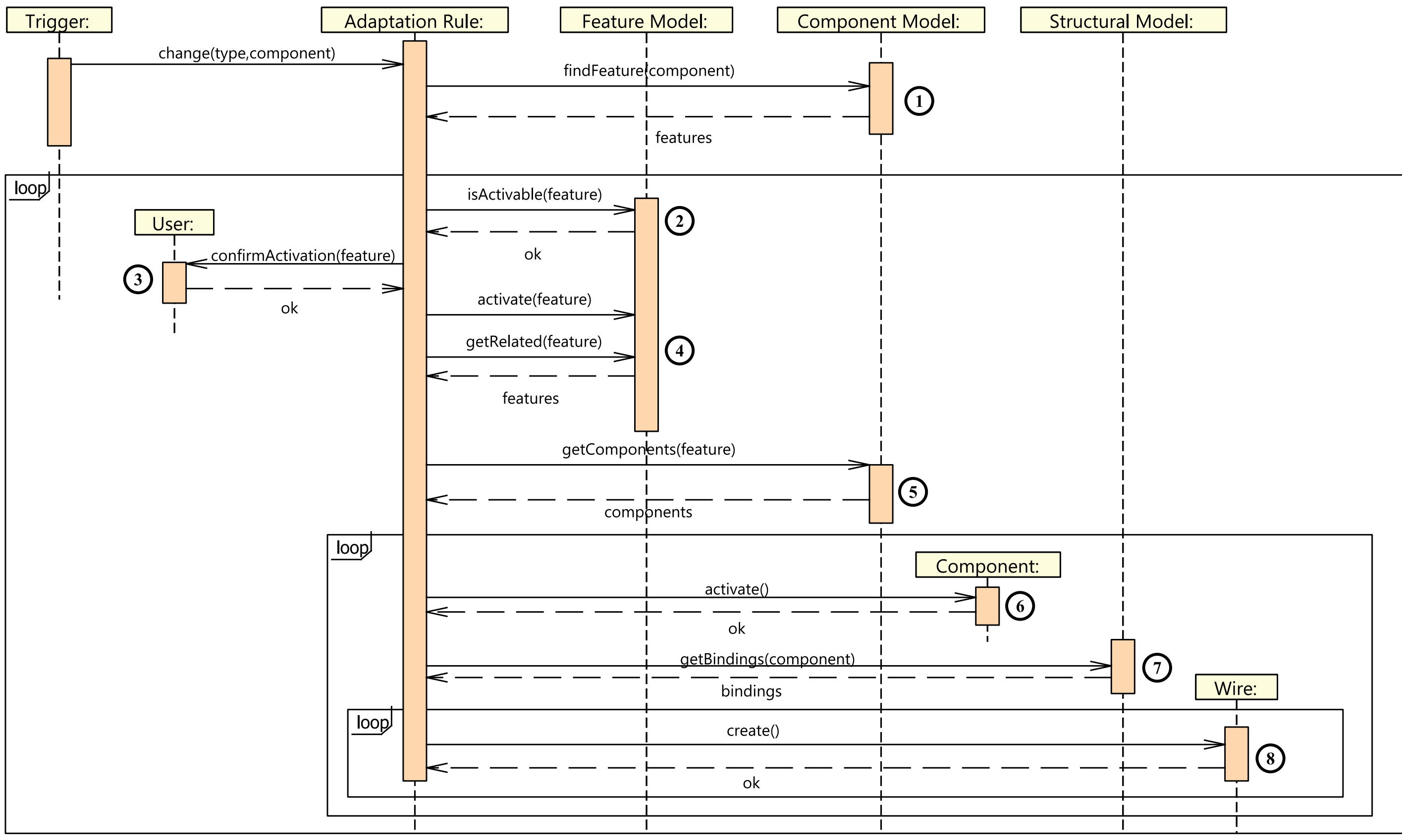

Fig. 2. Adaptation process for evolution scenarios.

1. By means of the Component model, the adaptation rule identifies those features which are related to the trigger component. With these features, the rule creates an ordered set called the *evolution set*. For each one of the features, the rule performs the following steps, 2 to 5.
2. The rule checks the possibility of feature activation. This information is in the Feature Model, specifically it depends on the requires, excludes and mandatory relationships between features. If all these constraints are fulfilled, then the feature can be activated.
3. Once the rule checks the feature activation, it asks the user for confirmation by means of a dialog in the user interface. The message shows the name of the feature and a description stored in he Feature Model. The message also provides three options to the user: "Yes", "Remind me later" and "No".
4. Activating a new feature can fulfil other feature constraints. In this step, the rule checks for new activable features. The rule adds these new features to the *evolution set*.
5. In terms of the platform, activating a feature implies performing *adaptation actions* to system components. In this step, the rule queries the Component model for the feature components. For each one of these components, the rule performs the following steps, 6 and 7.
6. The rule applies the *State-Shift* action to the component. Therefore, the component transits from a quiescent state to an active state.
7. To connect the new active component with the rest of the system, the rule queries the Structural model for the component bindings.
8. Finally, the rule applies the *Binding* action to create the communication channels between the components.

Due to space constraints, the sequence diagrams in this section represent only the general case for adaptation. Diagrams consider only affirmative responses, lacking alternative behaviour.

In our experience applying this approach to the smart home domain [2], we have notice that the time response delay comes mainly from these factors: feature dependency resolution (steps 2 and 4) and user confirmation (step 3). How much time the user takes to confirm cannot be foreseen, and dependency resolution is more time consuming than other simpler queries (for example, step 7). However, we consider that installing new resources in the system is not as critical as handling resource failures. Thus, in evolution scenarios we offer an advance system response (dependency resolution and user participation) although this response takes extra time.

4.2 Adaptation in Involution Scenario

Involution scenarios are triggered by the removal of a resource. A fast adaptation of the system is required to minimize the impact of the lost resource. In order to offer a good response time, adaptation is automatic (not requiring user intervention) and resource alternatives are precalculated in a model (the realization model). In this way, the latency of asking the user is avoided and the effort of reasoning with the feature model (e.g., looking for dependencies) is also reduced.

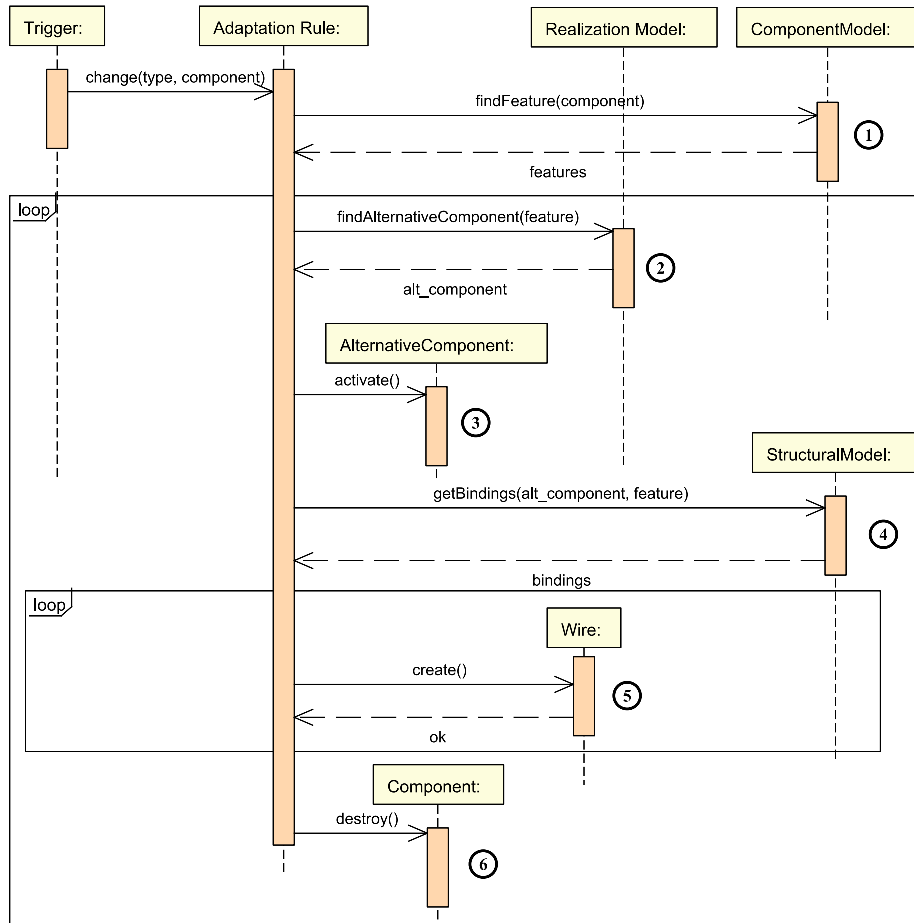

Fig. 3. Adaptation process for involution scenarios.

In Fig. 3, the adaptation process for an involution scenario is illustrated. Given the removal of a component, the affected feature is obtained and an alternative component for this feature can be directly retrieved from the Realization Model. More detail about the process is given below:

1. When a change is produced in the system, the affected features are obtained in the same way as in the evolution scenario. The following steps are performed for each feature.
2. The rule queries the Realization model to obtain a component that can replace the affected one for a given feature. Since this information is expressed explicitly in this model, queries are straightforward.
3. Once the rule has found an alternative component (initially in the quiescent state) it is activated.

4. The alternative component may require communication with other components. This information is obtained from the Structural model.
5. For each of the required bindings, a wire is created to establish the necessary communication channel between components.
6. Finally, the affected component is destroyed. This implies the removal of inactive wires. The destruction of this component is deferred until the end of the adaptation process, since the priority in involution scenarios is to offer the new services immediately.

The adaptation rule for involution reduces the delays commented for the evolution scenarios. On the one hand, model queries are simplified. Reasoning over a feature model is a time-consuming activity and termination becomes difficult to guarantee [9]. On the other hand, the user does not participate in the process, which is a requirement for the autonomic behavior required by this kind of scenarios.

5 Related Work

Hallsteinsen et al present the Madam approach [10] for adaptive systems. This approach builds systems as component based systems families with the variability modeled explicitly as part of the family architecture. Madam uses property annotations on components to describe their Quality of Service. For example a Video Streaming component may have properties such as start up time, jitter and frame drop. At run-time, the adaptation is performed using these properties and a utility function for selecting the component that best fits the current context.

Trinidad et al [11] present a process to automatically build a component model from a feature model based on the assumption that a feature can be modeled as a component. By means of augmenting the system with a feature model and a model reasoner, this approach enables systems to dynamically changing its features at run-time.

Both Hallsteinsen and Trinidad apply SPL techniques to develop adaptive systems. However, their approaches do not take into account the differences between evolution and involution scenarios. Therefore, they do not exploit the specific scenario requirements.

Zhang et al. [12] introduce a method for constructing and verifying adaptation models using Petri nets. They address directly the verifications of the adaptation models by means of visual inspection and automated analysis. On the other hand, our approach is focused on reuse at run-time the variability modeling of SPLs. However, our approach can benefit from SPL reasoners in order to check system properties [9]. Finally, Zhang's approach separates the adaptation specification and non adaptation specifications as our approach does. However, our approach introduce precalculated adaptations in order to achieve a faster response in involution scenarios.

6 Conclusions

In this paper, we provide support for adaptation in pervasive systems by means of run-time models. Our approach focusses on addressing the differences between evolution (a resource is added) and involution (a resource is removed) scenarios. In involution scenarios, we use models with precalculated knowledge in order to provide an autonomic response in a reduced amount of time. While in evolution scenarios, we offer an advanced system response (feature dependency resolution and user participation) because we consider that installing new resources in the system is not as critical as handling resource failures. Finally, we showed how models drive the system adaptation within the context of each scenario.

References

1. Weiser, M.: The computer for the 21st century. SIGMOBILE Mob. Comput. Commun. Rev. (3) (1999) 94–104
2. Cetina, C., Fons, J., Pelechano, V.: Applying Software Product Lines to Build Autonomic Pervasive Systems. 12th International Software Product Line Conference, SPLC 2008. (2008)
3. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Proceedings of the Third Software Product Line Conference 2004, Springer, LNCS 3154 (2004) 266–282
4. Fabro, M.D.D., Bzivin, J., Valduriez, P.: Weaving models with the eclipse amw plugin. In: Eclipse Modeling Symposium. (2006)
5. Bencomo, N., Blair, G., France, R.: Model-driven software adaptation report on the workshop m-adapt at ecoop 2007. Object-Oriented Technology. ECOOP 2007 Workshop Reader (2008) 132–141 Springer, LNCS.
6. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. Software Engineering, IEEE Transactions on Software Engineering (1990) 1293–1306
7. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: a case study. Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on Configurable Distributed Architecture (1998) 91–100
8. Marples, D., Kriens, P.: The open services gateway initiative: an introductory overview. Communications Magazine, IEEE (12) (Dec 2001) 110–114
9. Benavides, D., Segura, S., Trinidad, P., Ruiz-Corts, A.: FAMA: Tooling a framework for the automated analysis of feature models. In: Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems. (2007)
10. Hallsteinsen, S., Stav, E., Solberg, A., Floch, J.: Using product line techniques to build adaptive systems. Software Product Line Conference, 2006 10th International (Aug. 2006) 21–24
11. Trinidad, P., , Ruiz-Cortés, A., na, J.P.: Mapping feature models onto component models to build dynamic software product lines. International Workshop on Dynamic Software Product Line (2007)
12. Zhang, J., Cheng, B.: Model-based development of dynamically adaptive software. In: ICSE '06: Proceedings of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 371–380

An Execution Platform for Extensible Runtime Models

Mario Sánchez*[1,2], Iván Barrero[1], Jorge Villalobos[1], and Dirk Deridder[2]

[1] Software Construction Group, Universidad de los Andes, Bogotá, Colombia
{mar-san1,iv-barre,jvillalo}@uniandes.edu.co,
[2] System and Software Engineering Lab, Vrije Universiteit Brussels, Belgium
Dirk.Deridder@vub.ac.be

Abstract. In model-driven development, high level models are produced in the analysis or design phases, and then they are transformed and refined up to the implementation phase. The output of the last step usually includes executable code because it needs to introduce the details that are required for execution. However, some explicit structural information is lost or scattered, making it difficult to use information from the high level models to control and monitor the execution of the systems.

In this paper we propose the usage of a platform based on extensible, executable models, which alleviates the loss of information. When these models are used, the similarity between the structure of high level models and of the elements in runtime eases the construction and usage of the system. Moreover, it becomes possible to build reusable monitoring and control tools that are only dependent on the platform, and not on the specific applications. Our proposal is shown in the specific context of workflow-based applications, where monitoring and control is critical.

1 Introduction

In many applications, runtime information is necessary for a variety of reasons. For instance, it might be because it is necessary to take swift corrective actions during execution, or because historical data is required to check and improve performance. Runtime information might be difficult to manage and interpret, and thus it is desirable to have powerful tools to handle it. Furthermore, since such tools are difficult to build and maintain, then it is desirable to have reusable tools that can serve to monitor and control different applications.

Additionally, these tools should also be capable of managing high-level concepts. Modern systems allow business experts to have more direct control over the applications, instead of relying on technology experts as in the past. Because of this, it is expected for those applications to offer information in terms of high level business concepts. Similarly, the control interfaces should offer high level

* Supported by the VLIR funded CARAMELOS project http://ssel.vub.ac.be/caramelos/ and by Colciencias

operations instead of only low-level operations that require technical knowledge about implementation details.

One possible alternative to manage these requirements, is to build model-based tools to do runtime monitoring and control of applications. Given the flexibility offered by unsing models, such tools would be reusable and have the ability to manage high level concepts and information. However, the feasibility of building such tools depends on features required in the monitored applications. As we will see in this paper, these features are not always available.

Many platforms used to execute applications today have limitations. In the first place, the interfaces offered to gather runtime information or to control the execution of the applications are nor standardized; they do not even have commonalities among several applications. Thus, it is difficult to have reusable monitoring tools that manage information beyond the virtual machine or the operating system level. Furthermore, the interfaces are limited in the quality of information offered, which can be insufficient to adequately control the execution.

Another problem is that runtime information might be difficult to interpret in terms of business concepts. This happens because the mapping from implementation elements to high level concepts can be difficult to establish. As shown in figure 1, the analysis and design artifacts, which contain all the relevant business information, are somehow used to produce implementation artifacts. Depending on the strategy used, this step can produce different kinds of traceability information that can be later used to reconstruct the transformations. This figure also shows that monitoring and control tools are between the business and the implementation level. The functionalities of these tools depend on low level implementation data, events, notifications, actions, etc. However, the users that use the tools expect to see all those low level elements in terms of the corresponding business concepts. How to make this translation is thus the problem that has to be solved.

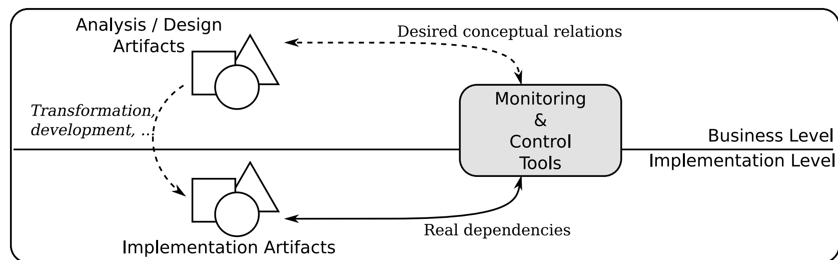

Fig. 1. Dependencies of monitoring tools.

One example of the previous ideas could be an application where *employees* have to perform a certain number of *activities*, and someone wants to monitor the percentage of activities finished by each employee in a given period. In this case, employee and activity are very precise business concepts, but they can be scat-

tered in the implementation. For instance, to retrieve the required performance information it might be necessary to get low level information about sessions, services invoked, database accesses and other. One alternative to preserve this information could be to keep traceability information. However, there are other kind of problems associated to the complexity of keeping track, managing and interpreting this information.

The proposal presented in this paper targets the creation of platforms that support the execution of a wide range of applications and offers the necessary features to develop reusable monitoring and control tools with the requirements discussed previously. In the first place, this proposal advocates for the development of model-based applications. In second place, it proposes a reusable platform for model execution, where runtime information is easily obtainable. Furthermore, since the elements in execution have a structure that is very close to the structure of elements in design, then it is easy to establish a mapping between implementation and business elements.

This paper focuses on presenting the Cumbia platform. This platform is based on extensible, executable models and it offers several advantages to runtime monitoring and control of the applications that are executed on it.

Nowadays, workflows and wokflow-based applications are very important and new ones are built permanently. Furthermore, in this context monitoring and controlling are critical. In this paper, the Cumbia platform is illustrated in the workflow context, but its advantages are also explained for more general contexts.

The structure of the paper is as follows. Section 2 presents the problem of monitoring and controlling runtime systems and, in particular, workflows. Afterwards, section 3 presents the details of our proposal, and section 4 shows how it eases the runtime monitoring, management and adaptation of model-based systems. Finally, some related works and the conclusions are presented.

2 Runtime monitoring and workflows

As we discussed previously, in many applications it is necessary to have access to runtime information that can be used, for example, to support runtime decision-making. Additionally, to have useful and powerful monitoring and control tools it is necessary to have the means to raise the level of abstraction of the available runtime information. This raise allows users of the tools to make analysis and decisions from the business perspective instead than from the implementation point of view. For instance, a non technical user might prefer to know that the level of service offered by a partner application has dropped below the limit specified in a contract, instead of having to understand reports about timeouts or communication failures.

Part of the complexity associated to raising the level of abstraction is to be found in the implementation of the applications. In many cases, applications have structures and architectures that are very different from the structure of the problems that are supposed to be modeled and solved. Thus, it is difficult to reconcile low level runtime information with high level concepts, as required

by monitoring and control tools. This problem, as well as the limitations in the interfaces to gather low-level information, becomes even more critical when monitoring requirements appear late in the life-cycle of the applications.

Based on the promises made by approaches like MDA [1] or xUML [2], model-based applications should face less problems to accommodate monitoring and control tools. However, in these approaches not all the discussed problems are solved because the last transformation or compilation step has as output executable code, which consistently scatters or loses some information that was originally available on the models. One approach that can be applied to overcome this problem inolves the usage of transformation models and traceabiliy information [3]. Although this might solve the inmediate problem of the lack of information, it creates other problems related to the interpretation of the information.

In the context of workflows and workflow-based applications these problems are also present. In the first place, in this context models are widely used by business experts, which use domain-specific languages to describe business processes taking into account the so-called business rules. Afterwards, these processes are deployed into workflow engines to be executed. At runtime, process designers expect to see the same concepts that they used in the definition. In order to make decisions, they have rules and policies that are based on that kind of information.

Normally, the execution of the processes is monitored and controlled either with low-level applications, such as engines' consoles, or with tools such as BAMs (Business Activity Monitoring). In general, BAMs are rather flexible and configurable, and they have as main goal raising the level of abstraction of execution information in order to make real time measurements of certain Key Performance Indicators (KPI) described by domain experts. However, BAMs are also tightly coupled to workflow engines' implementation, and to the workflow definition languages. As a result, the same BAM cannot be used with different engines. Moreover, even small changes to an engines' implementation, or to a language, can render unusable an entire BAM. This is something critical in this context because business rules and processes tend to evolve rapidly and the tools have to evolve along.

Finally, an increasingly important requirement in workflow applications is the ability to handle dynamic adaptation of the processes. This means that depending on internal or external events, the structure of processes might change at runtime, either in an autonomic fashion or following instructions specified by business experts. These changes have an impact on monitoring because the tools have to adapt and reflect the modifications. Furthermore, the tools used to describe the modifications should be integrated with the tools to do the monitoring and they should share the same high level language and concepts. It is expected that whoever directs the dynamic adaptation uses runtime information to make the necessary decisions.

3 Extensible executable models

The proposal presented in this paper is part of the Cumbia project of the Software Construction group of the Universidad de los Andes. In this project, we have developed the Cumbia platform for extensible, executable models. Originally, we developed this platform with the main goal of supporting extensible workflow-based applications, which might include complex monitoring requirements. Nevertheless, the platform is sufficiently generic and offers benefits that can be valuable in other contexts as well.

From the point of view of monitoring and control, a very important feature of the platform is the usage of executable models, which keep during execution all the structural information of the models. Since there is no loss of information, it is easier to rise the level of abstraction; in this case, creating the mapping between implementation and design elements is trivial. Another advantage of the platform is that it offers runtime information about every object in the model, that can be easily consulted from external applications. Moreover, the platform offers interfaces that can be used to easily integrate other applications, such as monitoring tools.

In order to support the execution of models, the platform manages the corresponding metamodels. The platform is very flexible in the support for metamodels, and also in the support for changes to the metamodels: they can grow and evolve without any significant impact to the platform. In section 3.1 it will be shown that the only requirement for metamodels is that they should be defined in terms of *open objects*, which are the special kind of executable elements that we defined for our platform. In this section, we will first briefly present XPM (eXtensible Process Metamodel), which is a simple metamodel for the definition of workflow processes that can be used in our platform. Afterwards, the main details about open objects will be discussed with the goal of explaining how monitoring and control are supported. More details about open objects can be found in [4].

3.1 XPM and Open Objects

The purpose of this section is to present one sample metamodel we have developed for the platform; we do not pretend to argue here about the completeness of the metamodel or its suitability to represent workflow processes. In order to present XPM we will use the sample process shown in figure 2, which was taken from the context of workflows for financial services. It defines a sequence of steps to study and approve a credit request. This process is initiated when a customer applies for a credit. Then, it requires an in-depth study of the submitted request and of the financial history of the customer. Finally, someone has to approve or reject the request based on the results generated by both studies.

This particular process shows most elements of the XPM metamodel. The process is composed by four *activities* that are connected through *ports* and *dataflows*. Each activity has a distinct *workspace* and each workspace executes

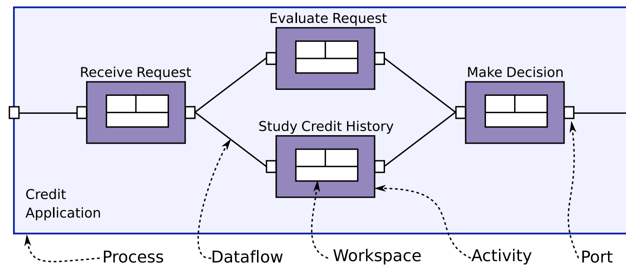

Fig. 2. A sample XPM process

a specific atomic task; activities serve to enclose them and handle all the synchronization and data management issues.

Every metamodel in Cumbia is based on something that we have called open objects, and thus they are the base for our platform. Every metamodel that is to be executed in our platform, has to be defined using open objects as its building block. As it will be shown, this means that every element in the metamodel has to be defined as a specialized open object. The workflow engine that we use to execute XPM processes was built using this approach and it has the open objects platform at its base.

The fundamental characteristic of an open object is that it is formed by an *entity*, a *state machine* associated to the entity, and a set of *actions*. An entity is just a traditional object with attributes and methods. It provides an attribute-based state to the open object and in its methods part of its behavior can be implemented. The state machine materializes an abstraction of the life-cycle of the entity, allowing other elements to know its state and react to its changes. Finally, actions are pieces of behavior that are associated to transitions of the state machine. When a transition is taken, its actions are executed in a synchronized way.

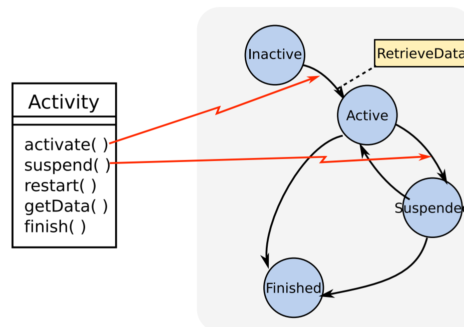

Fig. 3. The specialized open object that represents XPM activities.

The interaction between open objects is based on two mechanisms: event passing and method calling. In the specification of a state machine, each transition has an associated source event: when that event is received by the open object, that particular transition must be taken. This mechanism not only serves to synchronize open objects, but also serves to keep the state machine consistent with the internal-state of the entity: each time the latter is modified, it generates an event that changes the state in the automaton. Finally, other events are generated when transitions are taken and it is thus possible to synchronize open objects using state changes. The other interaction mechanism, method calling, is synchronous and is used to invoke the methods offered by the entities of open objects. These entities can receive method calls from two sources: there can be calls coming from external sources, such as user interaction or other related applications; additionally, the actions associated to transitions usually invoke methods of other entities, and thus they play a central role in the coordination of the entire model.

Figure 3 shows an open object that has been specialized to behave as an XPM activity. It has four states, and the state machine changes of state because of events generated by the entity or by other open objects. For instance it goes from the state `Inactive` to the state `Active` whenever the method `activate( )` is called, which in turn generates the event that moves the state machine. In addition, when this transition is taken, the action called `RetrieveData` is executed.

Several features of the platform and of the open objects facilitate the interaction with monitoring and control tools. In the first place, the platform is metamodel-based and changes to the metamodel can be seamlessly supported. This makes the platform particularly suitable to handle applications used in rapidly evolving contexts. In addition, open objects expose their state through the state machines, and this is an advantage for monitoring because the amount of available information. Furthermore, the interfaces provided by the platform offer powerful ways of interaction from external applications: it is possible to capture events to receive notifications, and it is also possible to invoke methods of the entities to control them (see figure 4). Another big advantage is that the mechanism of actions offers something similar to the interception meta-space described for the Lancaster Open ORB project [5, 6]: since actions can be installed and removed at runtime, it is possible to introduce extra-behavior between the processing of interactions. This additional behavior can be used to add further interactions with the monitoring applications. Finally, another big advantage offered by the platform is the inspection interface that allows to navigate the complete structure of the models, from the root level element (a process in XPM) to the last action or event.

Besides building the XPM engine, we have developed other metamodels and their corresponding “engines”. For instance, we have built the metamodels and the engines for BPMN [7] and BPEL [8]. In order to use the platform for them, the first step was to design the metamodels and implement the required open object specializations. Since the platform itself is responsible for managing the instantiation and execution of the models and their open objects, then some

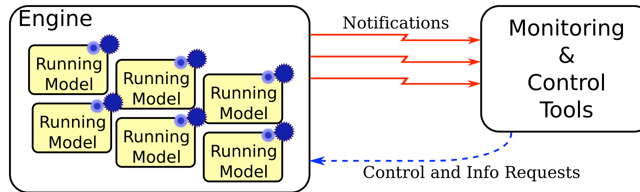

Fig. 4. Monitoring runtime models.

other specific services had to be defined for the engines in an ad-hoc way. For instance, in the case of the BPEL engine, the web-services based interface had to be specially developed.

4 Monitoring and control

By taking advantage of the features of the platform described, it is possible to build very powerful and reusable monitoring applications. The combination of using explicit metamodels and the existence of a single executable platform leads to monitoring tools that are highly configurable. Since these applications are only dependent on the platform, they can be easily adapted for the usage with systems based on other metamodels that can be executed in the platform.

When using the platform, monitoring tools can manage and offer four kinds of information about the running applications. The first kind represents the structure of the models, which normally is fairly static. The second kind is information about the current state of the elements in the models. The third kind is historical information about the state of the elements in the model, that is the trace or the history of the execution. In these three cases, we are dealing with elements available in the model, and accordingly to what was said previously, this means that we are dealing with high level concepts that are put into execution. Similarly, although all the notifications received from the platform are low level, they could be transformed and interpreted as high level notifications. For instance, a notification about a transition taken in the state machine of an activity, might be transformed into a high level notification about the completion of the activity.

The fourth and last kind of information that can be monitored is composed by derived information, which is not directly part of the model or its execution, but can be calculated. This derived information has to be defined for each context, including the rules necessary to calculate it using the information provided by the platform. To define, manage and analyze this kind of information, it is useful to have model-based monitoring tools, where the definition of the relevant information can be easily made. In the context of workflows, a possible example of derived information would be the average time required to execute the activities of a process. Another example, would be the name of the employee that performed more activities in a given month.

We have developed some examples of applications that monitor the execution of applications based in open objects by managing the first three kinds of information described. The most basic of those applications is an open objects viewer. For a given open object, this application allows the observation of the structure of the state machine and shows its state changes.

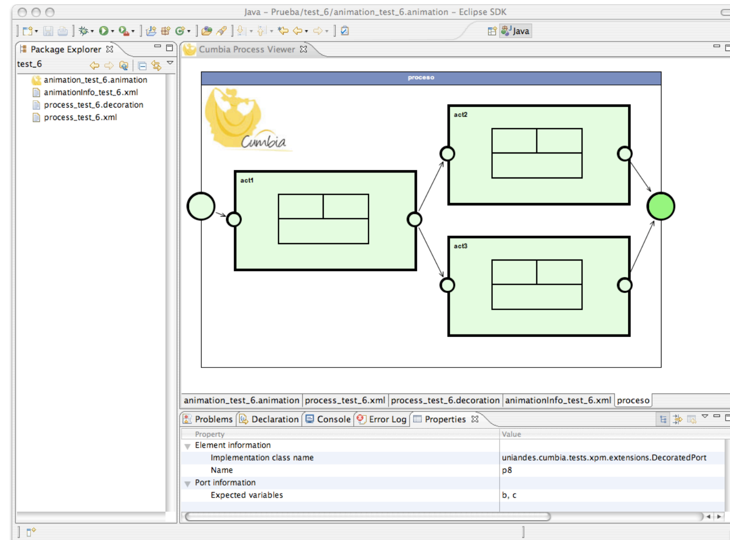

Fig. 5. Screenshot of the XPM Viewer.

Another application that we have developed is a viewer for XPM processes (see figure 5). This application not only shows the structure of the processes that are running inside the XPM engine, but also animates the elements shown by changing the color of the activities that are executed. This application has other two interesting characteristics. The first one is the possibility of using a domain specific language that describes what the viewer has to do when it receives notifications about state changes in XPM elements. Although the language itself is not very powerful (it only allows some basic stuff like changing the color figures based on types and state changes), it gives some flexibility to the viewer and turns it into an example of a simple configurable monitoring tool for open objects.

The second characteristic is that this application was developed as decoupled from the XPM engine as possible. As a result, the XPM engine ignores totally the existence of the viewer, while the viewer only has dependencies towards the open objects platform and the XPM metamodel (not the XPM engine).

The most complex monitoring application that we have built for the platform is called the “Cumbia Test Framework” (CTF). Although users do not interact with the tool, this application observes the execution of the models and interacts with them following the instructions in a script. Afterwards, the CTF

observes the execution, receives notifications and analyzes the traces to validate assertions. Moreover, the control statements in the script are described using a high level language. We have used the CTF to test the implementation of several metamodels, and in each case, the required specializations to the CTF have been minimal.

5 Conclusions

In this paper we have discussed about the importance of runtime monitoring and control and we have identified some useful requirements for monitoring applications. Among these requirements, the one that creates most of the implementation problems, is the need of giving high level information to the users of the tools, instead of providing implementation level information. Other problems that we explored were the limitations on the quantity and quality of the available runtime information, and the limited possibility of reuse for monitoring tools.

The proposal that we presented in this paper has two parts. First, it advocates for the use of model-based development techniques. Then, it proposes the usage of a platform based on extensible, executable models. This proposal has several advantages that, in the paper, were illustrated in the context of workflows. Among these advantages there is the usage of explicit metamodels, and the ease of integration with other applications besides control and monitoring tools.

The various benefits offered by the proposed platform are useful for the construction of monitoring and control tools. In particular, it makes possible the development of new, reusable monitoring tools that can be applied to several contexts which are to be executed on the platform.

References

1. Joaquin Miller and Jishnu Mukerji. MDA Guide. Object Management Group, Inc., Version 1.0.1., June 2003.
2. Mellor, S., Balcer, M.: Executable UML A Foundation For Model-Driven Architecture. Addison-Wesley, Indianapolis (2002).
3. Jouault, F.: Loosely Coupled Traceability for ATL In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany (2005).
4. Sánchez, M., Villalobos, J.: A flexible architecture to build workflows using aspect-oriented concepts. In: Workshop Aspect Oriented Modeling (Twelfth Edition), Belgium (2008)
5. Blair, G. S., Costa, F. M., Saikosky, K., and Clarke, N. P. H. D. M. (2001). The design and implementation of Open ORB version 2. IEEE Distributed Systems Online Journal, 2(6).
6. Costa, F. M., Provensi, L. L. and Vaz, F. F.: Towards a More Effective Coupling of Reflection and Runtime Metamodels for Middleware. In: Proceedings of Models@Run.time 2006, Genova, Italy (2006).
7. BPMN Information, http://www.bpmn.org/
8. BPEL Specification, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

Model-driven Management of Complex Systems

Brian Pickering[1], Sylvain Robert[2], Stéphane Ménoret[3], Erhan Mengusoglu[1]

[1] Pervasive & Advanced Messaging Technologies
IBM UK Laboratories, Hursley Park, Winchester, UK
[2] CEA LIST, Boîte courrier 65, Gif-sur-Yvette cedex, F-91191, Fr
[3] Thales Research and Technology, RD128, 91767 Palaiseau cedex, Fr
{brian_pickering, mengusog}@uk.ibm.com, sylvain.robert@cea.fr,
stephane.menoret@fr.thalesgroup.com

Abstract. This paper considers some specific issues relating to model-driven system management applied to complex systems. Examining dynamically coupled systems-of-systems on the one hand and highly distributed devices for service access on the other, we define a common meta-model of (semi-) automated management applicable in both domains. Taking monitoring by way of illustration, we then show how this meta-model is put into practice along two complementary aspects: management modelling and runtime event processing support.

Keywords: system management, runtime modelling, complex event processing.

1 Introduction

Previous work has looked at exploiting design-time architectural models at runtime in order to evaluate and validate potential changes to the current managed system [12], [8] and [9]. Although well motivated, because of power limitations [9] in the managed devices, or to check that potential changes would be of optimal use in the current environment [7] and so forth, there are issues about what can and cannot be captured at design-time. Kodase *et al* [10] suggest non-functional requirements are difficult to capture in design-time models, for instance; and Ulbrich *et al* [13] propose that quality-of-service management can only effectively be handled by message path manipulation during operation. In addition, most work from the seminal Oreizy *et al* paper on self-adaptation 11] to Rainbow [8] and the MADAM proposals for (mobile) telecommunication services [7] focus only on runtime modification of the managed system itself. There are, therefore, a number of important gaps here. Can we, for instance, introduce non-functional as well as functional aspects into our design time models? And can we now address omissions such as historical data, domain-specific rules and policy management (see the *Future Work* section in [7] for instance)? Most significantly, perhaps, can we make dynamic, context adaptive changes to the system management components using a design-time model just as we would for the managed system? This work attempts to address some of the issues raised in current model-driven approaches to system management. In this work, we seek to address

some of these questions. To begin with, we introduce our autonomic approach to system and device management (Section 2) and present the top-level meta-model we are defining for system management. Next, we consider specific issues related to management and system runtime modelling (Section 3), and finally (Section 4) we consider how to allow for runtime adaptation of the management system itself and the introduction of dynamically changing functional as well as non-functional requirements.

2 Model-Based Management Common Framework

Within the context of the MODELPLEX project, we have begun to define and evaluate common models for system management [1]. Although sharing some features with other approaches, notably OASIS, SMDS and a number of OMG initiatives *op.cit.*, we have focused our work on common issues which affect different aspects of system management. One specific area of interest involves the automatic and semi-automatic management of complex systems.

For system management purposes, we recognize a number of key concepts as summarized in Fig 1. A *ManageableElement* is the central and fundamental object which may be defined as any and all elements within a system that need to be managed. Each element is associated with one or more *ManageabilityCapabilities* that describe what and how that element needs to and can be managed. The elements, though, are not confined to those concepts and objects which are subject to being monitored and controlled. We must also include elements of the management system themselves, as well as the definitions of the criteria and rules by which the system is managed. So we need to be aware that *ManageableElements* may well include *ManageableSystemElements* or *ManagementRules* respectively.

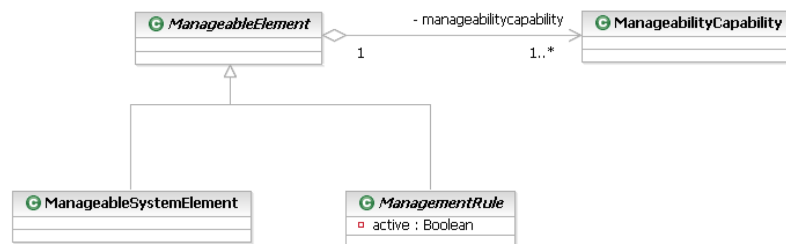

Fig 1: Top level common model for managing systems

From these central concepts, and in line with the work done by the Autonomic Computing Initiative (ACI) [2], we are in the process of evaluating the applicability of the MAPE-K autonomic management control loop. This provides for *monitoring,* data *analysis*, change *planning* and then *execution* of that plan on the basis of static and dynamic system *knowledge*, hence the acronym MAPE-K. We have extended the top-level model above with those concepts that relate specific to this approach.

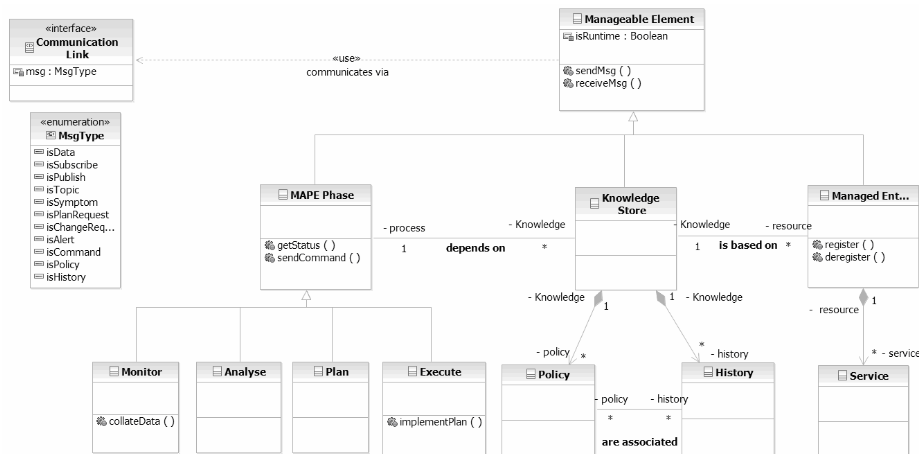

Fig 2: Design-time model of the MAPE-K control loop

Fig 2 illustrates a design-time model of the MAPE-K system management processes. Within the context of the MODELPLEX project, it derives from the common meta-models for system management described above. The MAPE management phases are themselves seen as *ManageableElements* as defined within the common meta-model [3], in much the same way as the *KnowledgeStore* itself containing both static data, or management *Policies*, and dynamic data from the *ManagedEntities*, held as *History*. These objects – *Policies* and *History* – are of particular interest at runtime in providing some way of potentially modifying management behaviours. Our challenge in evaluating such a model for system management lies in how it needs to be implemented, not least to establish whether the data objects can be changed effectively at runtime. This may provide a mechanism for changing the design-time management model. We need to consider further whether other factors need to be examined as well for a truly adaptive management operation.

3 Modelling Management and System Runtime

As far as management modelling is concerned, the MAPE-K loop and the common management meta-models (section 2) provide a conceptual basis. The current challenge is to develop a concrete expression for this framework. This requires defining precisely how models will be used in management, what shape they should take and how they relate to the design process. This section discusses these issues and sketches an experimental system management demonstrator for the monitor and analysis phases that we are implementing. We will begin in this section to consider the initial management phases (monitor and analyze) as they relate to specific aspects of System of Systems (SoS) management.

Enabling model-based system management obviously requires at least two features from the management models. First, since they are used by management operators to

observe and analyze system execution, they need to provide a runtime image of the managed system. Beyond that, it should contain management processing information, i.e. define how runtime information is handled (the Monitoring and Analyze phases of the MAPE loop) and – which is beyond the scope of this discussion – how corrective actions are deduced from this information (the Plan and Execute phases of the MAPE loop). On top of these base capabilities, management models should enable more complex management actions, such as determining the root cause of a runtime event (root cause analysis). Ideally, it should also be possible to act not only on the system but also on the management layer itself, i.e. an action on a management rule in the model would result in an actual action on the system management software. Finally, a model-based management tool suite should also allow the consequences of changes on the system or on the management system to be determined before they are performed.

Provided the tool support is adequate, the management concepts we have defined (Fig 3) meet these desired features in theory:

- To begin with, *ManageableElements* represent the monitored system elements. They own collected data (raw runtime data) and indicators (complex monitoring data built from processed collected data). Indicators can be processed to produce Symptoms, which are expressions of a departure from normal SoS function (note that symptoms are not related to any specific *ManageableElement*). Monitoring system execution thus comes down to observing *ManageableElements*, *CollectedData* and Indicators.
- Then, *ManagementRules* contain the management information: *MonitoringRules* express how *CollectedData* is processed to produce Indicators (logical and / or algebraic expressions the operands for which include *CollectedData*) and *AnalysisRules* express how *Indicators* are processed to produce *Symptoms*.

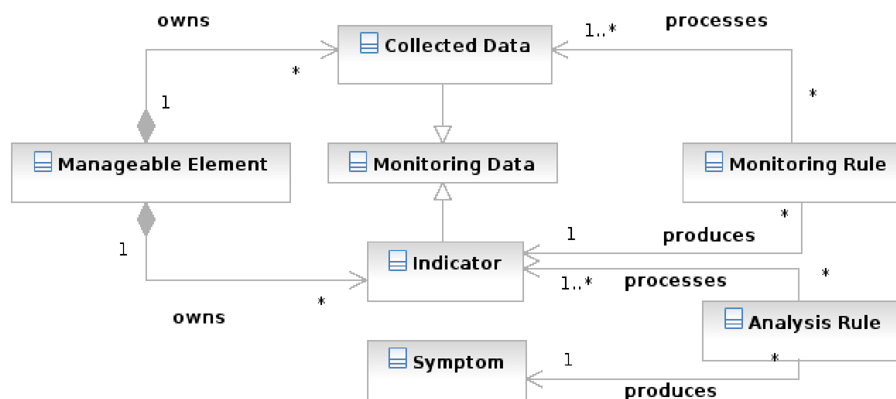

Fig 3: Some of the Management Meta-Model Concepts

However, as usual, concepts need to be put into to practice. In this respect, two concerns prevail: first, the modelling language (i.e. the concrete syntax associated with the meta-models); and subsequently, the tool support. As far as concrete syntax is concerned, it is possible to build runtime models by customizing design models (for instance in UML by creating and using a profile dedicated to management modelling). But this has one major drawback: the resulting models would provide too much detail compared to what is needed for management purposes, and that would reduce readability. We thus chose a different approach. Since we work in the scope of architecture frameworks, we propose to define a specific management viewpoint which will contain the management models. Doing so, these will not be confused with system models for other viewpoints. However, relationships will be defined between them in order to show how *ManageableElements* relate to actual system elements. For the definition of a management modelling language, we considered two equivalent options: either build a UML profile or define a DSL. We had favoured a DSL-based solution in order to enable fast and iterative prototyping.

In order to validate our choices, we have designed an initial experiment: a basic system management prototype which focuses on availability monitoring and analysis. The considered system is a set of cameras whose temperatures and states are monitored. The simulation scenario introduces runtime events (state and temperature changes) from which the monitoring / analysis chain would infer a faulty situation. One of the main outcomes of this prototype was the definition of a DSL for management modelling which gives a concrete shape to the concepts previously described (e.g. symptoms). This DSL was defined in the Microsoft DSL tool, which provides facilities to define the DSL meta-model and the associated concrete syntax and generates the corresponding domain-specific modelling environment. By way of illustration, Fig 4 shows an excerpt of a model of the system described above, built thanks to this DSL. This excerpt represents a *ManageableElement* associated with a camera (*Camera 1 ME*) owning collected data (*StateCD1* and *TempCD1*) and indicators (*S1*, *T1*), and a monitoring rule (*CDFilteringRule1*) which processes *TempCD1* to issue *T1*.

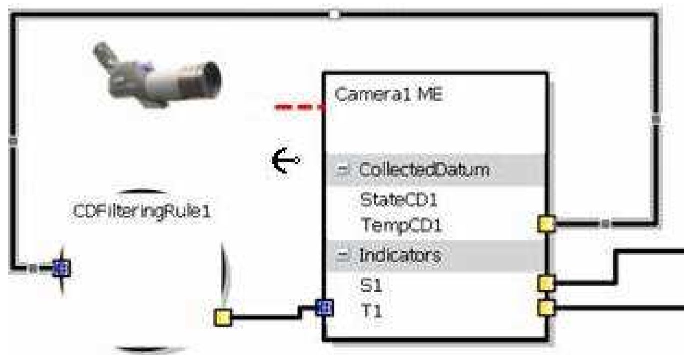

Fig 4: Excerpt of a management DSL model

In order to test our conceptual assets further, we also associated a C# script to each of the monitoring and analysis rules implementing their behaviour. Each time

incoming data values change (e.g. in the case of *CDFilteringRule1*, *TempCD1* changes), the script is executed and output data is issued (e.g. *T1* indicator for *CDFiltering1*). Messages are also displayed when events of interest arise (e.g. when a symptom is raised). In this way, the progress of monitoring and analysis can be observed directly on the model. To complete our experiment, a service-oriented Java application was implemented, which simulates the cameras and processes the scenario. This simulation performs dynamic updates of the model in the Microsoft DSL repository, which in turn trigger the model-level monitoring / analysis chain described above. These initial results are quite encouraging, since they demonstrate the global feasibility of our approach on a basic example. We now plan to extend the DSL, in order to enable more complex configurations, e.g. for layered management. We also envisage making some connections with more monitoring infrastructures, like the one described in the next section.

4 Complex Events Processing Architecture

The MAPE-K control loop introduced above provides a useful and extensible design-time model for the semi and fully automatic management of systems or devices. Pickering et al [5] have begun to assess its applicability to highly-distributed devices within a service provider network. In this section, we will focus on the runtime modelling of the monitoring phase specifically.

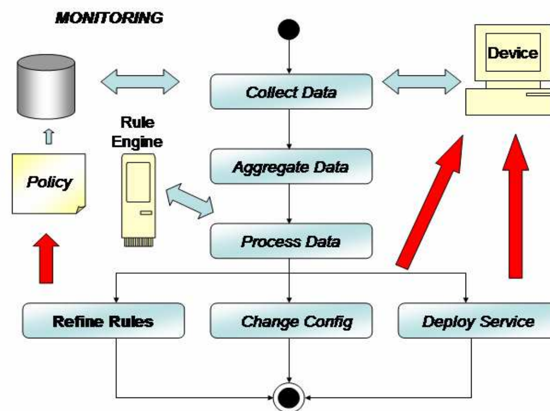

Fig 5: Monitoring Activities at Runtime

Our initial approach to autonomic computing via the MAPE-K framework would suggest a unidirectional process flow for management. Monitoring data are retrieved from the services or devices managed during the initial phase. These are processed and evaluated using policy data from the knowledge store by the analysis and plan phases. The results from these are then sent to the execution phase to effect either are reconfiguration of the managed system or the deployment of a new or modified service. This basic unidirectional flow from monitoring data collection based on

policies (*ManagementRules* or service level agreements) to configuration change or service deployment (via the *Execution*) phase is shown in Fig 5.

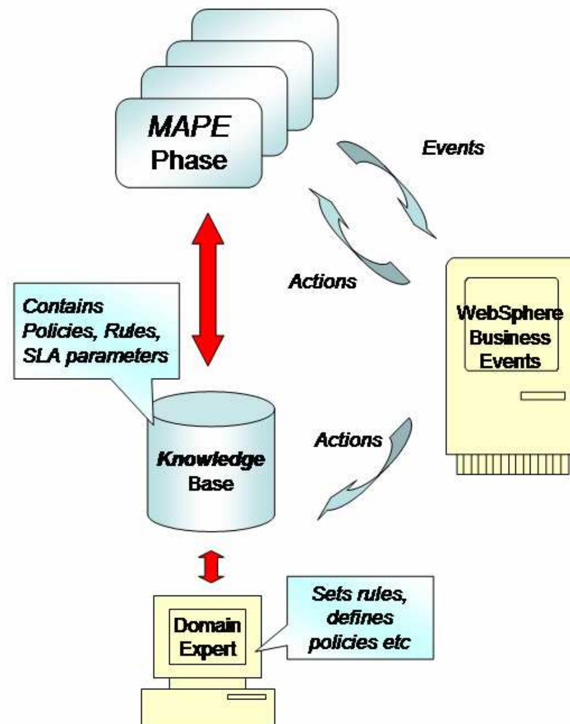

Fig 6: MAPE-K and Event Processing

The *Policies* can be regarded as the set of domain-specific operations applied to the data obtained by the monitoring process. We use a *rule engine* for complex event processing to aggregate and analyze the data and then make inferences to decide what to do next. The IBM WebSphere® Business Events platform is the best candidate for this rule engine in our architecture since it provides both simple and complex event processing (CEP). We might begin by seeing the rule engine in this case assuming the role of analysis and plan phases in the MAPE-K approach. This preserves the cyclical nature associated with control loops: monitor (or *observe*), decide and take action; then return to the monitor step. In addition to the execution of simple rules related to relatively simple events, such as threshold checking for instance, the rule engine needs to be able to detect complex event patterns in order to provide a complete monitoring process in terms of data aggregation. Complex events may be defined in IBM WebSphere Business Events as rules about the co-occurrence or order of events, but may also be extended to use additional event data for the definition of correlation patterns. With this complex filtering, we do well to reconsider whether the rule engine does fulfill the function of the *Analysis* and *Plan* phases in MAPE-K.

Fig 6 summarises how we use the rule engine within our MAPE-K implementation. Irrespective of MAPE phase, the management process retrieves operational rules (policies, SLA parameters, management rules) from the Knowledge Store at runtime. But also, as data are handled, *MonitoringData* in the case of the monitoring phase, then the rule engine is presented with the data (via an *Event*) to correlate in accordance with the event filters, which as stated may be simple or complex. The result of this processing may result in a change to the operational rules (signaled via an *Action* event to the Knowledge Store); this introduces dynamic data and rule control for our system management model.

In practice, we don't see the sort of uniform behaviour whereby data from a service are monitored, undergo initial processing and are then passed on to the next management phase. Suppose for instance that circumstances change. For instance, rerouting delivery across the network may affect the service provider's ability to meet agreed throughput levels. Such changes will result from the managed system. Equally, some changes may be commercially motivated: customer status or service features may change with a knock-on effect for policy handling. The changes are externally motivated, and independent of the managed system itself. We must therefore allow feedback about the policies and rules from the management system back into the knowledge store from a number of sources. This can be handled as outlined above and summarised in Fig 6: the operational rules are modified by IBM WebSphere Business Events via an *Action* to the Knowledge Store. This in turn may modify how the monitoring is done. As such, the process flow must include a non-device-affecting path back to the knowledge store as well as to the MAPE phases themselves. In Fig 6, *Actions* may therefore return to the MAPE phase as well as to the Knowledge Store. Such *Actions* may be directives (phase I/O parameters or configuration settings), which may include, of course, the next MAPE phase to be called, if any. Our control loop flow is therefore bi-directional. We are able, therefore, to modify how we process the management data from the managed system, but also how the management system itself operates at runtime. Such dynamic changes are reflected back as temporary or permanent modifications to our design-time system management model.

5 Related Works

In section 3, we focused on issues related to runtime system and monitoring operations modeling. This work is based on the common meta-models introduced in section 2 and as such builds on the results of [2] for autonomic management. Our contribution also sits well amidst earlier works on architecture-based system management, like [8], or works about models simulation like [17]. As far as management modelling support is concerned, we are also deriving some benefit from works within domain-specific modelling, such as [16]. Since runtime information layout is one of our concerns, we also have a connection with work such as [15] about dynamic models layouts, though our scope is far more comprehensive. Turning to Section 4, we considered issues pertaining to operational models for system management. Continuing work on autonomic management presented by González *et*

al [6], we took the ACI MAPE-K framework [2] as our starting point, and more specifically the monitoring phase. Using complex event processing (CEP) [14], we have been able to introduce elements of dynamism to the management system itself; we are now free to generate modified managed system configurations at runtime in contrast to the preloading proposed by Illner *et al* [9] for the service provider domain. In addition, instead of relying on the fixed design-time model of our management control loop, we are also able to introduce changes to the management system itself, and not just adaptation to apply to the managed system along (see [7], 11] and so forth). Further, by viewing management policies, SLAs, and monitoring parameters as dynamic data which can be modified at runtime in response to some CEP-type filtering. Integrating multiple, dynamic data sources of these types introduces the concepts Floch *et al* [7] call for with MADAM.

6 Conclusion

This paper has presented an approach to model-based management for complex systems with a focus on two adjacent aspects. The first is modelling support for management, which entails both model-level visualization of the running system as well as the model-based definition of management functions; and the second is runtime support for complex event processing. On the first aspect, we have proposed a viewpoint dedicated to management concerns. This viewpoint enables – thanks to a dedicated domain-specific language – both monitoring and analysis rules which specify the management logic as well as a runtime view of the system as a set of so-called "manageable elements" to be modelled. This view is then continuously updated as system execution progresses. The second point we deal with relates to runtime management support. We provide an infrastructure based on IBM WebSphere which performs complex and basic runtime event processing (i.e. processes the monitoring and analyze management chain). In accordance with the conceptual model presented in section 2, runtime events can concern the system itself and – something which is not that usual - the management rules themselves. This infrastructure thus permits the terms of management to be modified at runtime.

The added value of our work mainly lies in its comprehensiveness, since we aim at providing support for the whole management chain, from its model-based specification to its realization. Moreover, our strict MDD stance – we clearly place models at the foreground of the development process, both for management specification and system representation at runtime – is not very widespread for such management facilities. Finally, the way we propose to act on the management itself (i.e. to manage the management) at runtime can also be regarded as an original contribution to the field.

On top of implementation and experimentation issues, the next steps will deal with, the improvement of our modelling support. In particular, we plan to enhance the management domain-specific language to enable hierarchical monitoring data processing. This requires in particular the definition of aggregation mechanisms for high-level management indicators (indicators, symptoms), which we have not considered yet. In managing the management system, we are also planning to

examine further the implications of distributed management: how to maintain currency or some level of versioning between the original design-time, centralized models and those adapted at runtime; and how and under what circumstances we can distribute complex event processing across the hierarchical network topologies of typical service-providers.

Acknowledgments. The work presented in this paper is being carried out in the context of the MODELPLEX project (IST-FP6-2006 Contract No. 34081), co-funded by the European Commission as part of the 6th Framework Programme.

References

1. Model-based systems management state of the art, MODELPLEX deliverable D5.1.a, MODELPLEX project, 2007.
2. Autonomic Computing, http://www.ibm.com/autonomic/
3. Common meta-models for system management, MODELPLEX deliverable D5.1b, MODELPLEX project, 2007
4. Griffen, C, Huang, R B, Sen, Z, and Fiammante, M "Tranforming UML <<Activity>> Diagrams to WebSphere Business Modeler processes" 2007 http://www.ibm.com/developerworks/websphere/techjournal/0707_fiammante/0707_fiammante.html
5. Pickering, B, Fernández, M A, Castillo, A, Mengusoglu, E "A Domain-Specific Modelling Approach for Autonomic Network Management" 2008 MACE
6. González, J M, Lozano, J A, López de Vergara, J E and Villagrá, V A "Self-adapted Service Offering for Residential Environments" 2007
7. Floch, J, Hallsteinsen, S, Stab,m E, Eliassen, F, Lund, K and Gjørven, E. "Using Architecture Models for Runtime Adaptability" 2006 IEEE Software
8. Garlan, D, Cheng, S-W, Huang, A-C, Schmerl, B and Steenkiste, P "Rainbow: Architecture Based Self-Adaptation with Reusable Infrastructure" 2004 Computer
9. Illner, S, Pohl, A, Krumm, H, Lück, I, Manka, D and Sparenberg, T "Automated Runtime Management of Embedded Service Systems Based on Design-Time Modeling and Model Transformation" 2005 INDIN
10. Kodase, S, Wang, S and Shin, K G "Transforming Structural Model to Runtime Model of Embedded Software with Real-time Constraints" 2003 DATE'03
11. Oreizy, P, Gorlick, M M, Taylor, R N, Heimbigner, D, Johnson, G, Medvidovic, N, Quilici, A, Rosenblum, D.S and Wolf A L "An Architecture-Based Approach to Self-Adaptive Software" 1999 IEEE Intelligent Systems
12. Poirot, P-E, Nogiec, J and Ren, S "A framework for constructing adaptive and reconfigurable systems" 2007 IEEE
13. Ulbrich, A, Weis, T and Geihs, K "QoS Mechanism Composition at Design-Time and Runtime" 2003 ICDCSW'03
14. IBM WebSphere Business Events http://publib.boulder.ibm.com/infocenter/wbevents/v6r1m0/index.jsp
15. S. Prochnow, R. von Hanxleden, "Enhanchements of Statecherts Modeling – the Kiel environment", Artist 2007, Berlin, Germany.
16. Kelly, S., Tolvanen, J.-P., "Domain-Specific Modeling", IEE Computer Society Publications, 2008.
17. Combemale, B., X. Crégut et al., Introducing Simulation and Model Animation in the MDE TopCased Toolkit, ERTS 2008, Toulouse, France.

K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines*

Brice Morin, Olivier Barais, and Jean-Marc Jézéquel

IRISA / INRIA Rennes / Université Rennes 1
EPI Triskell, Campus de Beaulieu
35042 Rennes Cedex, France

Abstract. Software systems should often provide continuous services and cannot easily be stopped. However, in order to meet new requirements from the user or the marketing, systems should be able to evolve in order to provide new services or modify existing ones. Adapting software systems at runtime is not an easy task and should be realized with attention. In this paper, we present K@RT, our generic and extensible framework for managing dynamic software product lines. K@RT is composed of three parts: *i)* a generic and extensible metamodel for describing running systems at a high-level of abstraction, *ii)* a set of meta-aspects that extends the generic metamodel with constraint checking, supervising and connections with execution platforms *iii)* some platform-specific causal connections that allow us to supervise systems running on different execution platforms.

1 Introduction

Developing, testing and validating adaptive systems is a daunting task. Indeed, such systems can propose a wide range of possible configurations at runtime [13, 17]. These systems can be seen as Dynamic Software Product Lines (DSPL) that can reconfigure themselves at runtime.

In order to facilitate the development, test and validation of DSPLs, we propose K@RT, our aspect-oriented and model-oriented framework for supervising component-based systems. This generic framework is independent from any underlying execution platform and proposes to maintain a reference model at runtime [6]. Using this high-level view of the running system, we can navigate the runtime architecture using model-oriented languages [19] and invoke services that are delegated to the running system. K@RT also allows to adapt the running system by modifying its runtime model, checking constraints on the modified model and comparing the actual reference model to the modified model. This process produces a safe reconfiguration script that is executed on the running system. The modified model may be obtained with high-level model-transformation languages [19] or Aspect-Oriented Modeling (AOM) approaches [11, 15, 17, 18], avoiding users to write low-level platform-specific reconfiguration scripts.

The remainder of this paper is organized as follows. Section 2 introduces our generic and extensible metamodel for representing models at runtime. Section 3 briefly presents

* This work was funded by the DiVA project (EU FP7 STREP, Theme 1.2: Service and Software Architectures, Infrastructures and engineering, Contract 215412)

our causal link between a running system and a runtime model. Section 4 details the aspect-oriented architecture of K@RT. Section 5 evaluates our framework. Finally, Section 6 concludes and outlines future works.

2 A Generic and Extensible Metamodel for Runtime Models

In this section, we present our generic metamodel[1] for representing component-based systems at runtime. This metamodel does not aim at representing high-level architectures but focuses on abstracting a running system. This metamodel is independent from any execution platform and can easily be mapped on Fractal [7, 8], OpenCOM [9], or SCA [1].

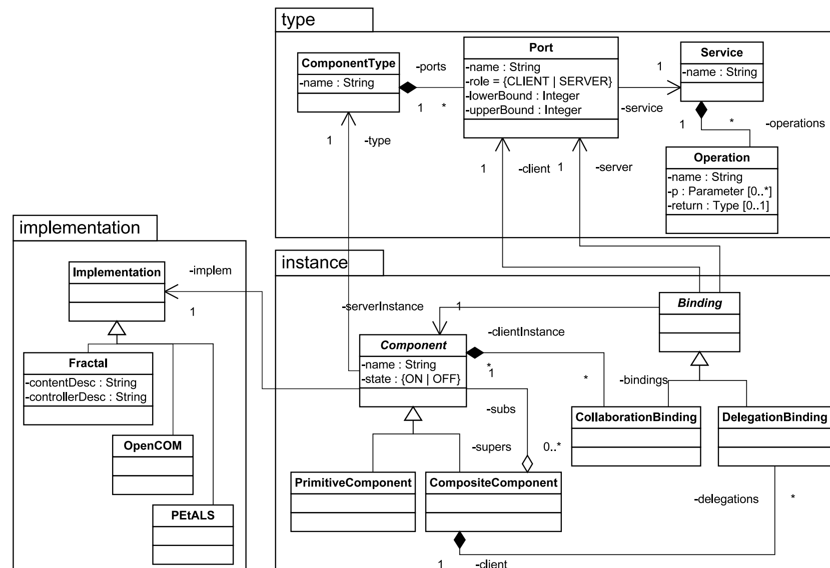

Fig. 1. A Generic and Extensible Metamodel

Our generic metamodel is separated in three packages, as illustrated in Figure 1. The **type** package defines the notion of component type. A component type contains some ports. Each port has a UML-like cardinality (upper and lower bounds) indicating if the port is optional (*lowerBound* == 0) or mandatory (*lowerBound* > 0). It also indicate if the port only allows single bindings (*upperBound* == 1) or multiple bindings (*upperBound* > 1). A port also declares a role (client or server) and is associated to a service. A service encapsulates some operations, defined by a name, a return type and some parameters. Basically, a service has a similar structure than a Java interface.

[1] In this paper, "metamodel" refers to the MOF terminology, not the middleware terminology.

The **instance** package defines the actual topology of a running system. A component has a type and a state (ON/OFF), specifying whether the component is started or stopped. It can be bound to other instances by a collaboration binding, linking a provided service (server port) to a required service (client port). A composite instance can additionally declare sub-instances and delegation bindings. Note that our metamodel allows shared components as a component may have several super components. A delegation binding specifies that a service from a sub-component is exported by the composite instance.

The **implementation** package contains metaclasses responsible for encapsulating the platform-specific attributes needed to implement components for a given platform. For example in Fractal, we should specify the implementation class (*contentDesc*) and a controller (*controllerDesc*) in order to be able to create a component.

We prefered to define a domain-specific metamodel (DSM) rather than reusing for example the UML 2.0 metamodel. Indeed, a reference model conforming to this metamodel is causally connected to the running system. Using a DSM allows us to reduce the number of entities that have to be maintained at runtime and consequently limit the memory overhead. This metamodel is strongly-typed and allows us to define algorithms with few casts whereas it is often necessary to perform casts when working at the platform level as they often deal with losely-typed objects. Moreover, this metamodel is aligned on the Service Component Architecture (SCA) [1] metamodel proposed by industrial partners like IBM, Sun, Oracle, SAP or Siemens. Our metamodel can be seen as a ligthweight version of SCA. This allows us to easily map our metamodel to SCA [1] and reuse the tools provided by SCA, such as a graphical editor to visualize the runtime architecture.

3 A Model-Driven Causal Connection

This section briefly presents our model-driven causal connection between a reference model, conforming to the metamodel we have presented in Section 2, and an execution platform. Currently, we have implemented such a causal connection for the Fractal [7, 8] platform but it can also be implemented for other component-based execution platforms like OpenCOM [9], if they provide reflection and dynamic reconfiguration mechanisms. The architecture of this causal connection is illustrated in Figure 2 and is detailed in the next two subsections.

The *Model2Platform* component is in charge of reflecting the changes of the model to the platform. This components will be detailed in this section. Identically, the *Platform2Model* component reflects the changes of the running system to the model. These two components use the *Factory* component in order to instantiate model elements from runtime entities, and vice-versa. The *Root* component is a composite component that contains the system designed by the user. This component is not really part of the causal link and may be deployed on a different site than the other components implementing the causal connection.

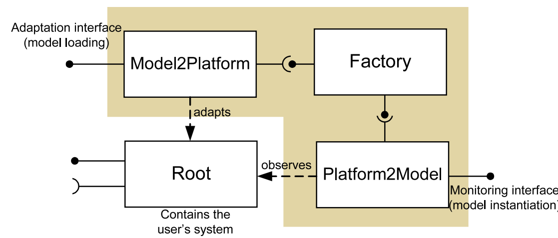

Fig. 2. Architecture of our Causal Connection

3.1 From Platform to Model

This subsection describes how we generate and update a reference model that represents, at a higher level of abstraction, the running system.

Fractal [7, 8] and all the reflective component-based execution platforms propose mechanisms for introspecting a running system. These mechanisms allows to discover which components actually compose the system, how they are bound to each others, etc. We extend the introspection operations provided by middleware approaches in order to discover the operations and their parameters that are provided/required by ports. In the Java-based distribution of Fractal or OpenCOM, each port (provided or required interface) is associated to a Java interface. We use the `java.lang.reflect` API to discover these operations and give a more precise view of the system.

Using reflection is very useful to instantiate a model from scratch. But, if we want to keep the model up-to-date, instantiating a complete model periodically may be time and resource consuming if only minor changes occurs. We have instrumented the Fractal platform to observe and notify all the architectural reconfigurations that appear in the running system. This allows us to update the reference model.

Finally, it is possible to visualize the runtime architecture in the graphical editor provided by SCA. Indeed, we have defined a model transformation in Kermeta [19], that maps the concepts of our metamodel to the concepts of SCA.

3.2 From Model to Platform

This subsection describes the other part of our causal connection. In K@RT, the only way to adapt a running system is to submit a new model to the causal link (see Section 4). When a new model is submitted to the causal link, we perform a difference analysis between the modified model and the actual reference model. In the current implementation of K@RT, we use EMF Compare [2] in order to realize this analysis. EMFCompare provides a generic comparison engine that can be customized for any domain-specific metamodel.

After analyzing the output provided by the comparison engine, we can determine what has been removed from the model, added into the model or updated. However, we cannot directly adapt the running system using these elements. Indeed, we cannot ensure that the order we discover the modifications during the analysis will result in

a consistent adaptation of the running system. For example, if we discover that some bindings and some components have been removed, it would probably lead to dangling bindings in the running system if we directly adapt the system. In order to adapt the running system in a consistent way, we reify every significant modification as a command. Each command declares a priority (*e.g.*, a command that removes a binding has a higher priority than a command that removes a component). These commands are automatically ordered with a Comparator. Once all the commands are instantiated, they are executed in the right order in order to actually adapt the running system. We first stop the components that needs to be stopped, we remove all the bindings and the components, add the new components and the bindings and finally restarts the components.

4 K@RT: Kermeta at RunTime

This section presents our aspect-oriented and model-oriented framework for supervising component-based systems at runtime. This framework is based on the generic and extensible metamodel presented in Section 2 (Figure 1) and is implemented in Kermeta [19]. Three Kermeta meta-aspects, **constraint checker**, **supervising** and **platform adapter** extends the generic metamodel, as illustrated in Figure 3. Kermeta meta-aspects allows us to statically introduce new features in existing model elements: adding classes in packages, adding super classes in the inheritance tree, adding and implementing new operations and adding contracts (invariants, pre/post conditions).

4.1 Constraint checker meta-aspect

This subsection details the constraint checker meta-aspect. This aspect weaves invariants into metaclasses. These invariants can be written in OCL [3] and translated into Kermeta thanks to the OCL Kermeta plugin, or directly written in Kermeta. We illustrate this aspect by detailing one of the invariants we have implemented.

The *completeCollaborationBindings* invariant illustrated in Figure 4 specifies that all the client (*PortRole.CLIENT*) and non optional ports defined in the type (*self.type*) of the component should be targeted (*b.client*) by the client reference of a binding owned by the component (*self.binding*).

This invariant uses the OCL-compliant operators provided by Kermeta (*e.g.* select, forAll, exists, etc), which significantly reduce the complexity of writing invariants. The same invariants implemented in Java/EMF needs 15 lines of code and would even be more complex if it was directly implemented using the platform API.

Specifying constraints on the metamodel allows us to check well-formedness rules that all the runtime models, and consequently all the running systems must respect. Using model-oriented constraint languages like OCL or Kermeta allows designers to rapidly implement such invariants as these languages propose high-level operators for manipulating models. Note that it is possible to define additional constraints in the constraint checker aspect. For example, if the underlying execution platform do not support shared component, an invariant can check that components have no more than one super component. Currently, 6 invariants are implemented in the constraint checker aspect.

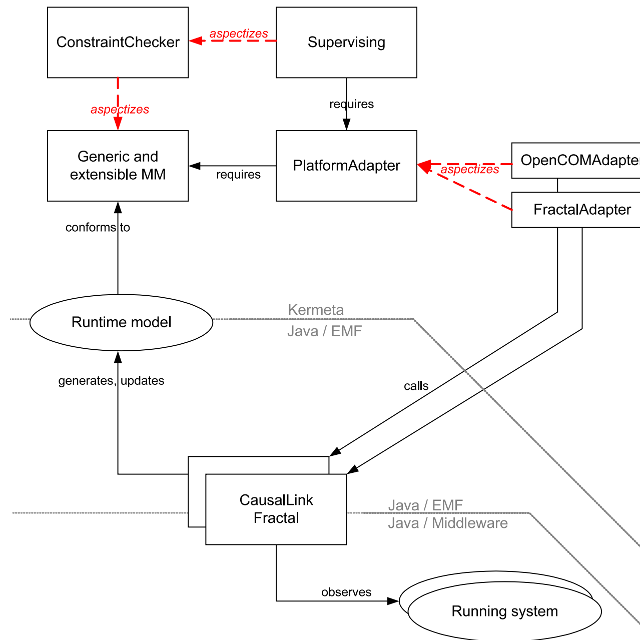

Fig. 3. K@RT overview

4.2 Supervising meta-aspect

The supervising aspect implements an administration console. It introduces two metaclasses: *DisplayContext* and *DisplayElement*. The *DisplayContext* metaclass is responsible for managing the history of the administration console and provides some useful method for displaying information. The *DisplayElement* simply defines an abstract operation *display(context : DisplayContext)*. In the aspect, this metaclass is introduced as a super class for all the elements that may be displayed: *Component*, *ComponentType*, *Binding*, etc. The *display* operation is implemented in each subclass. The *DisplayContext* and *DisplayElement* metaclasses can be seen as an interactive and history-aware visitor pattern allowing to display the elements chosen by the user and to go back to the previously visited elements.

4.3 Adapter meta-aspect

This aspect is responsible for connecting Kermeta to the execution platform. Kermeta proposes a seamless mechanisms for calling Java programs. Thus, it is possible to connect our K@RT framework with Java-based distribution of Fractal (Julia, AOKell), OpenCOM, etc. Currently, the Fractal adapter is fully functional and other adapters are under development. The adapter aspect proposes operations for:

```
1  aspect class Component {
2    inv completeCollaborationBindings is do
3      self.type.ports.select{p |
4        not p.isOptional and p.role == PortRole.CLIENT}
5          .forAll{p |self.binding.exists{b |
6            b.client == p}}
7    end
8  }
```

Fig. 4. Component metaclass aspectized with an invariant

- Instantiating the reference model from scratch using the introspection API provided by the underlying middleware platform. In Fractal, we use the content, binding, name, lifecycle and attribute controllers.
- Getting the current reference model using the notification mechanisms provided by the underlying middleware platform. This allows updating the reference model instead of generating it from scratch. We have implemented a new controller for Fractal that notifies all the runtime architectural changes to registered observers.
- Loading a model. It loads the model, analyzes the diff and match models and computes a safe reconfiguration script, as described in Section 3.
- Invoking services. Fractal does not propose controllers for easily accessing and invoking methods in a reflective way. We tackle this issue by directly using the `java.lang.reflect` API in order to discover which operations can be called and actually call them from Kermeta. We plan to integrate this implementation in a Fractal controller.

4.4 Discussion

K@RT is implemented according to our generic metamodel, instead of directly refering to an underlying execution platform *e.g.* Fractal, OpenCOM, etc. It allows us to reuse it for different platforms provided that they could be mapped, in both directions, to the metamodel. However, if the execution platform cannot directly be mapped to the metamodel, it is possible to aspectize the metamodel and the related meta-aspects to extend them with new concepts.

Fractal Explorer [4] is a tool for managing Fractal-based applications via a graphical console. Currently, our console is textual but it would be possible to connect K@RT to a Java-based graphical console, as Kermeta programs can be connected to Java programs. The main differences between Fractal Explorer and K@RT are summarized below:

- K@RT is technology independent while Fractal Explorer is based on Fractal. Indeed, K@RT is based on our generic and extensible metamodel that allows us to connect it to different execution platforms
- K@RT offers a higher level of abstraction. Indeed, in Fractal Explorer all the details of the Fractal component model are displayed in the console: content-controller,

binding-controller, lifecycle-controller, etc. In K@RT, a Fractal component that declares a binding-controller and a lifecycle-controller is simply represented by a component that contains some bindings and declares a state. The Fractal-specific notion of controller is abstracted.
– K@RT offers a higher level reconfiguration process. Indeed, K@RT proposes to adapt the running system by modifying the reference model. It is possible to use any transformation languages like Kermeta [19] or Aspect-Oriented Modeling tools [11, 15, 18] to modify the model. Then, this modified model is checked and automatically translated in a safe reconfiguration script. Fractal explorer is limited to fine grained reconfiguration.

5 Evaluation of K@RT

In order to evaluate K@RT, we have implemented a prototype in Fractal. This example is based on the service discovery system [12]. A service discovery system can either *advertise* or *request* services. It can also provide both functionalities. A service discovery system can communicate via several technologies (at least one). In this example, we propose WiFi and Bluetooth (BT). Figure 5 shows the complete architecture of the service discovery system, with both roles and both communication technologies. This model is automatically generated from the running system and mapped to SCA.

We define each functional role (Advertiser or Requester) as an aspect and each communication technology (WiFi or Bluetooth) as an aspect. We use an Aspect-Oriented Modeling tool (SmartAdapters [14–16]) to weave these aspects and produce all the possible configurations [15, 20]. There exists 9 possible configurations for the service discovery system: Advertiser role, Requester role or both and WiFi, Bluetooth or both. Consequently there is 9*(9-1) = 72 possible transitions from one configuration to another. Our causal link succeed to reconfigure the system at runtime for all these 72 transitions. The average time for reconfiguration was 200 ms.

Since all the aspects are independent from each others, it would be possible to handle the adaptive behavior of the service discovery systems with 8 scripts, for adding/removing each aspects. The identification of aspect dependencies and the generation of the minimal set of reconfiguration scripts will be subject to future work.

6 Conclusion and Future Works

In this paper, we have presented K@RT, our framework for developing, testing and validating Dynamic Software Product Lines (DSPL). This framework allows us to construct adaptive systems by defining model transformations [19] or weaving aspects into a base model [15, 17, 20]. It is possible to check the different configurations of the system, represented by platform independent models that can be visualized in a graphical editor. Using our causal link, it is possible to adapt a system at runtime and switch from one configuration to another, without writing reconfiguration scripts. The metamodel we use is generic and the tool is currently implemented for the Fractal platform. The mapping with OpenCOM may not be problematic because of the similarities between both component models.

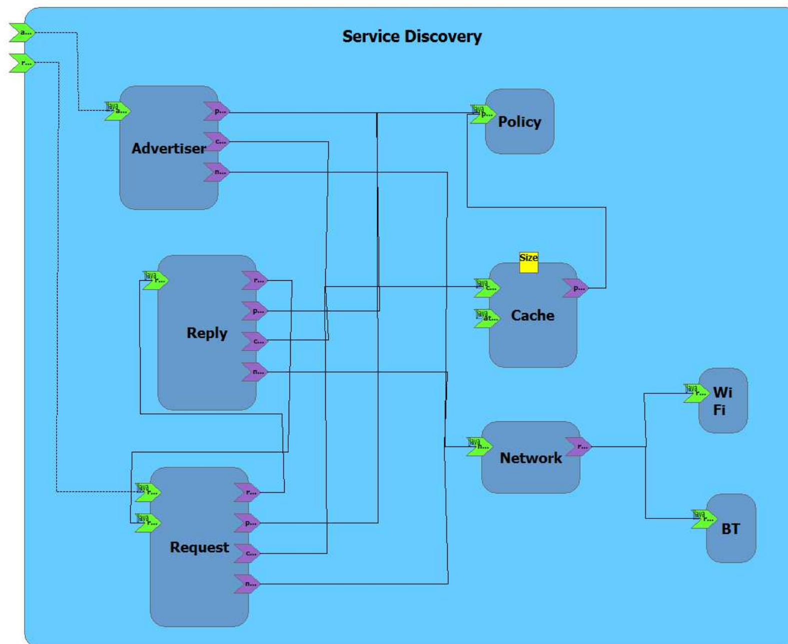

Fig. 5. Service Discovery Runtime Architecture

In future work, we plan to reuse an existing framework for monitoring interesting properties of the environment and develop a reasoning framework that will automatically select or generate (*e.g.*, by weaving some aspects into a base model [17]) the most adapted configuration. After checking some constraints, our causal link will automatically reconfigure the running system. We plan to use the WildCAT monitoring framework [10] in combination with the Intel Mobile Platform Software Development Kit [5] that provides a set of implemented probes. Another interesting future work would be to implement the mapping toward an OSGi platform (Equinox for example). Indeed, OSGi is largely used in the industry: mobile phone industry with Nokia, automotive industry with BMW. In the context of the DiVA project, our industrial partner CAS[2] proposes a Customer Relationship Management application based on Equinox/OSGi. This application would help us in testing and our tool in an industrial context.

Acknowledgment

Brice Morin thanks the *Collège Doctoral International* of the *Université Européenne de Bretagne* for funding his visit to Lancaster University.

[2] See http://www.cas.de/english/

References

1. Service Component Architecture http://www.eclipse.org/stp/sca/.
2. EMF Compare http://www.eclipse.org/emft/projects/compare/.
3. Object Constraint Language Specification, version 2.0 http://www.omg.org/technology/documents/formal/ocl.htm/.
4. Fractal Explorer http://fractal.ow2.org/fractalexplorer/.
5. Intel Mobile Platform Software Development Kit. http://ossmpsdk.intel.com/ and http://sourceforge.net/projects/mpsdk.
6. N. Bencomo, G. Blair, and R. France. Models@run.time (at MoDELS) workshops. http://www.comp.lancs.ac.uk/bencomo/MRT.
7. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. An Open Component Model and Its Support in Java. In *CBSE'04: 7th Int. Symp. on Component-based Software Engineering*, pages 7–22, 2004.
8. 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. Fractal is available at: http://fractal.ow2.org/.
9. G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas. The Design of a Configurable and Reconfigurable Middleware Platform. *Distrib. Comput.*, 15(2):109–126, 2002. OpenCOM is available at: http://sourceforge.net/projects/gridkit.
10. P.C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *SC'06: 5th Int. Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97, Vienna, Austria, 2006.
11. F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *AOM@MoDELS'07: 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.
12. C. Flores-Cortés, G. Blair, and P. Grace. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. *IEEE Dist. Systems Online*, 8(7):1, 2007.
13. S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4):93–95, 2008.
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. B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd Int. ECOOP'07 Workshop on Models and Aspects, Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.
17. B. Morin, F. Fleurey, N. Bencomo, J-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *MoDELS'08: 11th Int. Conf. on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.
18. B. Morin, J.Klein, O. Barais, and J. M. Jézéquel. A Generic Weaver for Supporting Product Lines. In *EA@ICSE'08: Int. Workshop on Early Aspects*, Leipzig, Germany, May 2008.
19. 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.
20. G. Perrouin, J. Klein, N. Guelfi, and J.M. Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *SPLC'08: 12th Int. Conf. on Software Product Lines*, 2008.