A Model-Driven Process for Self-Adaptive Software

Franck Chauvel, Isabelle Borne, Jean-Marc Jézéquel, Olivier Barais

A Model-Driven Process for Self-Adaptive Software

Franck Chauvel[1,2], Olivier Barais[2], Jean-Marc Jézéquel[2], Isabelle Borne[1]

1: VALORIA & University of South Brittany
Centre de recherche Yves Coppens
Campus de Tohannic
56017 VANNES CEDEX, France

2: IRISA & Université de Rennes 1
Campus de Beaulieu
F-35042 RENNES, FRANCE

Abstract: Many Embedded Systems are supposed to run continuously, which includes recovering from errors by adapting their configuration or their architecture to changing conditions in their environment. The design of such systems has to relate some high-level extra-functional properties to some low level ones such as memory or CPU consumption by defining some complex feed-back loops for the dynamic adaptation of the system.

However, the design of such feed-back loops (also called “adaptation policies”) is still a very complex endeavour if you want to go beyond predefined fall-back modes. Since the expression of extra-functional properties and the design of adaptation policies are complex activities, they are generally delayed down to implementation time. Adaptation policies are then implemented without either high level design nor dedicated tests, which may lead to costly roll-back operations in the design process.

To avoid such roll-back operations, we suggest a model-driven process based on new executable meta-modelling techniques. At modelling time, designers have to complement the architectural description with some sensors and actuators related to the involved extra-functional properties. It allows designers to specify in a consistent way the related adaptation policies. Then since the model is executable, some simulations of the adaptation policies can be performed at design time to evaluate their performances with respect to some relevant test scenarios. Then, using model-driven transformations, it allows the generation of code skeletons for real-time embedded platforms. In this article we illustrate our approach with a simple case study based on a mobile video player that is able to adapt its architecture to varying conditions, such as bandwidth evolution or low battery conditions.

Keywords: Component-Based Systems, UML, Self-Adaptation, Simulation, Code Generation.

1. Introduction

Embedded systems are designed to provide specific services rather that general purpose software. Most embedded systems are deployed on hardware that is expected to run continuously for many years and which needs to recover when errors occur. Designing such systems implies to describe the quality of the provided services according to the resources and the services provided by the environment. Specifications for embedded software typically entail high-level quality requirements specified by the user. While these are related to the application domain, they also often have low level performance implications. For example, a video-player might be required to provide acceptable quality (defined, e.g. in terms of frame rate and resolution) through some video playback. Such requirements can be maintained using adaptation policies that try to maximise the quality in response to the environment changes. While several approaches make it possible to relate high level extra-functional properties to low-level concerns [11, 12, 18, 9], none of them take into account adaptable component-based systems, where this relation is dynamic because it might depend on sophisticated adaptation policies introducing complex feedback loops by having a deep impact on the behaviour of some components or even on the component architecture itself.

In this paper we propose a tool supported approach allowing the development of such arbitrary complex adaptation policies. Our work is focused on component-based software architectures expressed using a subset of the UML2.0 notation [19] in which we enable the definition and simulation of adaptation policies. We propose to reify extra-functional concerns into so-called extra-functional components encapsulating both the monitoring of the relevant properties and the adaptation policies. Building on recent advances in executable meta-modelling techniques, these component-based architectures can be given operational semantics encompassing both functional and extra-functional aspects. This operational semantics then makes it possible to run simulations in order to get performance estimates with respect to some operational profiles.

The rest of the paper is organised as follows. Section 2 starts by introducing the motivating example of a hand-held wireless video player where the designers are interested in trying to maximise the

frame rate while the bandwidth changes. It then describes how this video player can be modelled and given an operational semantics, taking into account both functional and extra-functional aspects, including for its adaptation policies. Section 4 shows how this model can be used in conjunction with an operational profile to get early performance evaluation of competing adaptation policies. Section 4 discusses the advantages and the drawbacks of our approach whereas Section 5 presents a model transformation which enables the generation of code skeleton for the Fractal platform. Section 6 discusses related works and the paper finishes with some conclusions and perspectives.

2. Motivations and overview of the process

2.1 Motivations

We propose a wireless video player as a running example for illustrating the ideas presented in this paper. This video player's main function is to stream a video downloaded from a wireless network. Figure 1 presents a possible architecture for that kind of video player.

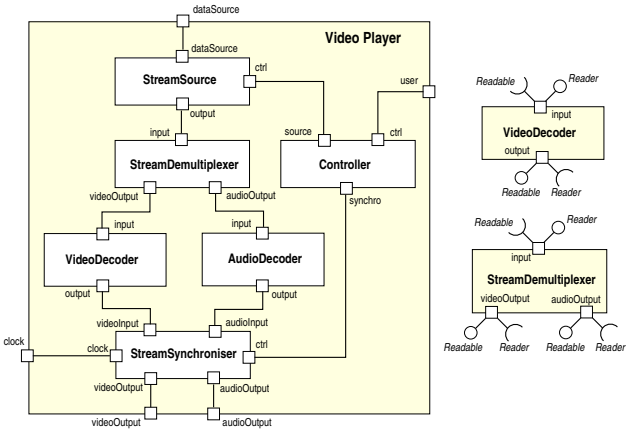

Figure 1 – A possible software architecture of a video player

Since video decoding is a data-oriented process, the video-player system is shown as a composite component which encapsulates a collaboration between several sub-components where each sub-component represents a step of the process.

1. A StreamSource component reads data from a file or from the network.
2. A Demultiplexer component separates the audio stream from the video stream.
3. Each stream is decoded by the relevant Decoder component.
4. A Synchroniser sends audio and video data to relevant renderers while managing the synchronisation between the two streams.
5. A Controller component propagates the user command to the process.

Since the video player component is dedicated for deployment on mobile devices where the environment might change quickly, designers need to include adaptation policies in order to maximise the quality of the provided service. According to the literature about multimedia streaming [8], the main user-perceived quality property is called the Frame Rate and corresponds to the number of video frames that the system is able to process per unit of time. If the frame rate is very high, then the movie seems to be fluent whereas if the frame rate falls, then the movie becomes jerky.

Under a highly dynamic environment, one of the parameter which can impact the frame rate is the bandwidth available for downloading the video. The contribution of this paper is to enable the development of the adaptation policy which ensure the a high frame rate while the bandwidth decreases by adapting the architecture of the system.

2.2 Overview of the process

The process we suggest is introduced by the figure 2. It divides the development of self-adaptive software into 6 steps which are described in the remainder of the paper.

1. The first step consists in building the architecture without any consideration for the adaptation policies. We use a subset of the UML notation, including component diagram for the structural view, activity diagrams and statecharts for the behavioural aspects, and class diagram to describe the data.
2. Then, the designer needs to complete the architecture with some probes which measures the extra-functional properties that he wants to involve in the adaptation policies. We use the same language (the subset of UML) to design the probes used in the architecture.
3. The third step is devoted to complete the design with the self-adaptation policies which drive the response of the system to changes in the system or in the environment. We use a rule-based language to describe adaptation policies.
4. Once the full self-adaptive system has been described, it must be validated in order to avoid cost effective rollback operations in the development process. We use a simulation-based approach which enables a high level and abstract simulation at design time to validate the adaptation policies.
5. When the self-adaptation policies have been validated, then the architectural model can be transformed into a platform model. This platform model reflects the specific

characteristics of a component-based platform such as CORBA or Fractal.

6. From this platform model, executable code skeletons can be generated. This code generation is mainly focussed on the structural view but the interfaces between probes, functional architecture and the self-adaptation policies are preserved. Moreover, the adaptation policy described by rules can be directly interpreted by the an *ad hoc* Fractal extension that we have developed.

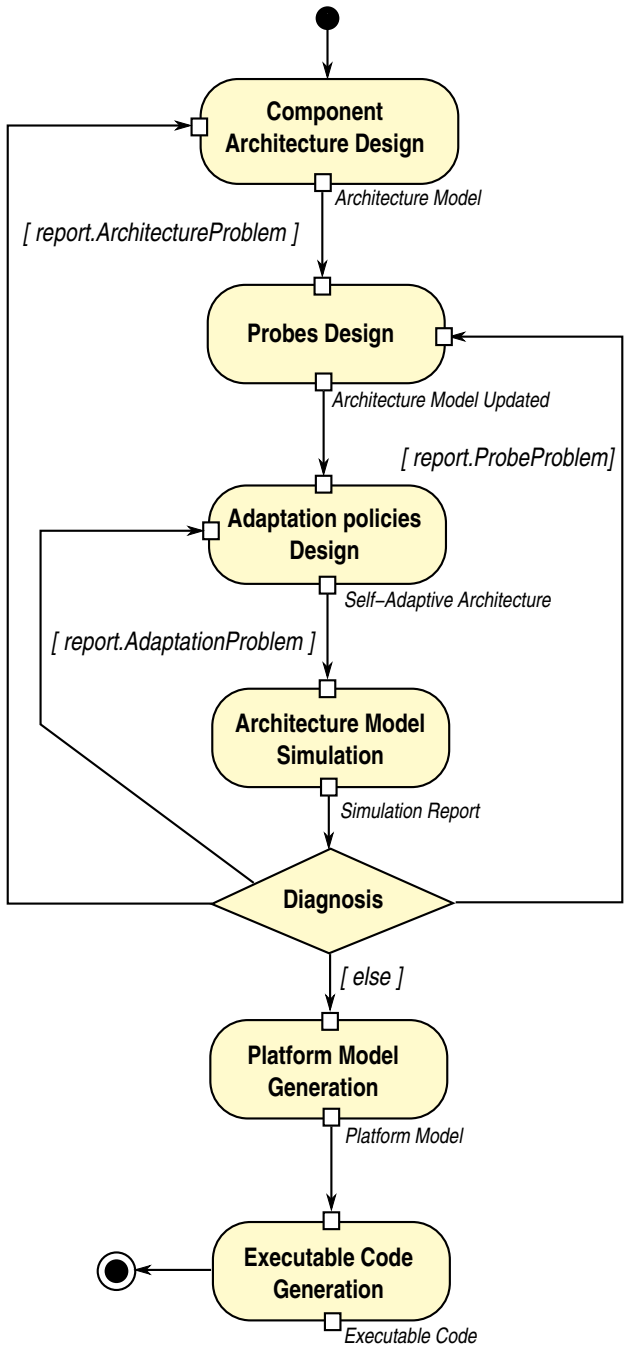

Figure 2 -The model driven process devoted to self-adaptive systems

3. Modelling Self-Adaptive Software

3.1 Modelling Component-Based Software

In order to be able to simulate component-based architectures, and thus to get performance results of adaptation policies, it is necessary to describe architecture in a precise and operational way. Components are described using three main points of view, namely the structural view, the behavioural view and the data view.

The **structural view** allows designers to group functionalities expressed as services on components. These components are the basic elements of the system and will be put together to build more complex collaborations encapsulated into composite components as shown in Figure 1. Figure 3 gives the structural view of the Demultiplexer component. It defines three ports, namely the input port which provides the data to process, the audioOutput port and the videoOutput port. Each port is typed by a set of provided or required interfaces which define the services provided (or required) by the port.

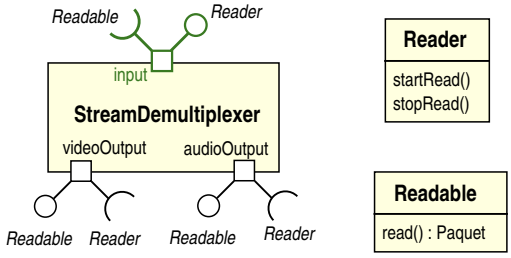

Figure 3 - The structural view of the Stream-Demultiplexer component depicted as a UML component diagram

The **data view** describes the data processed or stored by a component as a class diagram. These data might contain operations but only as passive objects, that is to say objects which cannot call any external features. Figure 4 presents the data used by Demultiplexer component as a class diagram attached to it. It defines 3 buffers: one for recording the data read on the input port, and two other ones for recording the audio and video data which have just been demultiplexed.

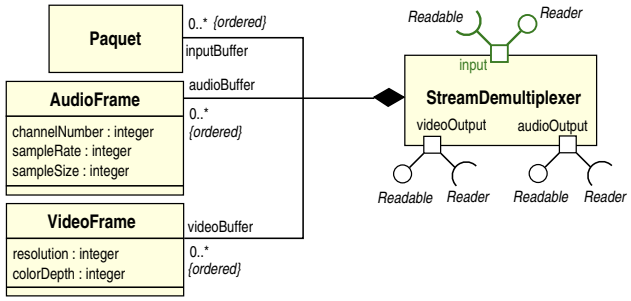

Figure 4 - The data view of the Stream-Demultiplexer component

The **behavioural view** enables the use of both statechart diagrams and activity diagrams to describe the different behaviours of a component. Statecharts are used to describe the coordination between the features that are provided by the component whereas activity diagrams are used to describe the internal use of the features that are required by the component. The link between statecharts and activities is set in both states and transitions. Transitions link an event to an activity whereas states might refer to an exit (or an entry) activity that will be performed by exiting (or entering) from the state. The goal of the behavioural view is to abstract away the functional processing performed by the component. Figure 5 shows an excerpt of the behavioural view of the Demultiplexer component. The left part presents the process which handles the reading of the data on the input port whereas the details of the activity which is performed when some new data have been read on this port.

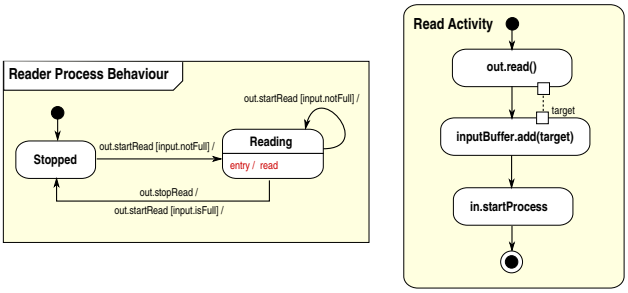

Figure 5 - The behavioural view of the StreamDemultiplexer component

3.2 Modelling probes for extra-functional properties

In order to model the behaviour related to self-adaptation, the designer must deal with extra-functional properties such as memory consumption for instance. These properties must be reified in the architecture as well as the others services which have been designed to provide functional services. For instance, if the designer wants to address the memory consumption, then he needs to design a service devoted to measure the current amount of free memory.

This reification of extra-functional properties as functional services can be boiled down to the development of specific probes (sensor and/or actuators). Such probes are included in a real self-adaptive system to measure extra-functional properties. However, the large amount of technical details required to design such probes fall out of the scope of early design activities. What the designer need is to reflect the expected behaviour of the probe.

We describe probes as basic components and then connect them into the functional architecture. In the video player example, since the objective is to maximise the frame rate with respect to the available bandwidth, both of these two quality properties need to be reified.

The bandwidth-related events are produced by an extra-functional component named Bandwidth-Monitor which watches the available bandwidth and calls the appropriate services when the bandwidth changes. The BandwidthMonitor component also provides a way to share of the available bandwidth. This functionality allows designers to express the bandwidth needs of the other components of the architecture as a service call (or a signal notification). Figure 6 gives some details of the component BandwidthMonitor. It offers three connection points, namely context, user and owner. The context port offers services that enable the configuration of the bandwidth (bandwidth available, thresholds, etc). The user ports enable other components to be connected and express their needs of bandwidth whereas the owner port enables the notification of the owner component of the bandwidth changes. The BandwidthMonitor component can store every bandwidth request that occurs.

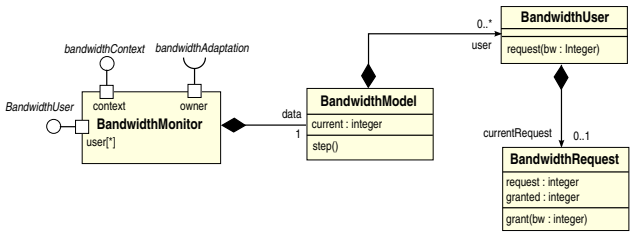

Figure 6 - A possible design for the bandwidth probe

In order to make the probes effective, the designer needs to insert them into the architectures. This kind of hand-crafted “weaving” is applied at two levels:

- At the structural level, the probes which appears as component are connected to the rest of the architecture through ports and connectors
- At the behavioural level, the designer needs to update the behaviour of each component in order to reflect the side effects of the functional behaviour on the extra-functional properties.

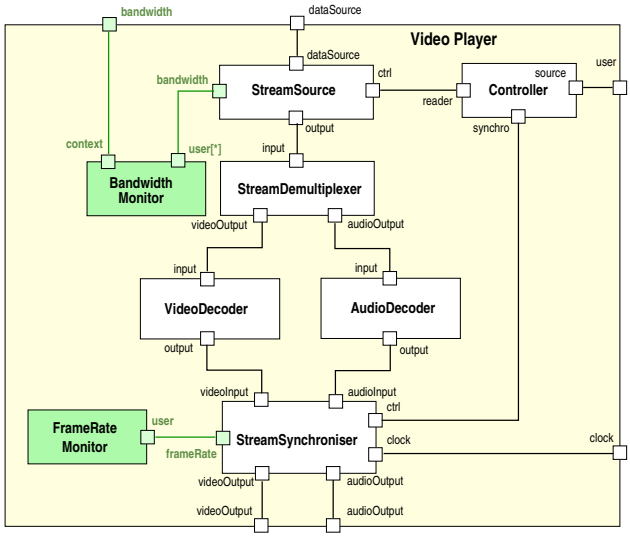

Figure 7 - The structural view of the video player component updated with the two probes components

Figure 7 shows the structural view of the video-player where two probes have been added. The first one (BandwidthMonitor) measures the bandwidth consumption and is connected to the StreamSource component. The second one (FrameRateMonitor) measures the frame rate and is connected to the StreamSynchronizer component.

On the behavioural view, the behaviours of both the StreamSource and StreamSynchronizer have been updated. Figure 8 shows the way we have completed the behavioural view. The ReadData activity of the StreamDemultiplexer component as been updated with two activities which computes the amount of bandwidth needed to read data on the network. The left part of the figure shows that the Synchronize activity of the FrameRateMonitor when a new frame is ready to be displayed.

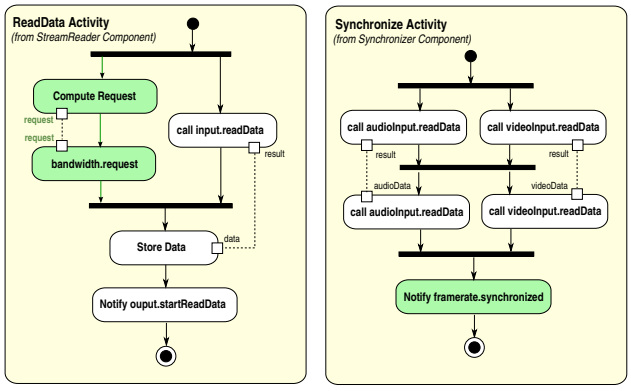

Figure 8 -The behavioural view updated with the side effects of the functional behaviour.

3.3 Modelling Self-Adaptation behaviour

Finally the designer must describe the self-adaptation behaviour, that is to say, the way he wants the system to respond to changes in the environment. In the video player for instance, the objective is to keep the frame rate as high as possible even if the bandwidth falls and this can be achieve by several fall-backs actions. First, the loss rate of the image encoding can be increased in agreement with the streaming server. Moreover, grey scale pictures can be used to save bandwidth if the bandwidth is still decreasing. Finally, if the bandwidth available is too low, then it is possible to dynamically change the component used for decoding. The system can switch from an MPEG2 to an MPEG4 data stream to save bandwidth.

The self-adaptation policies are described as the behaviour of composite components. A composite component (such as the video-player) encapsulates collaboration between several components and it must be in charge of maintaining the correct configuration of the collaboration. For instance, since the main objective of the video-player is to maintain the frame rate as high as possible with respect to the bandwidth evolution, the video-player itself (the composite component) is in charge of updating the architecture.

This self-adaptation policy is described using a rule-based notation where each rule describes how to react to event coming from probes or from other components. Events are mapped on asynchronous calls received by composite component.

```
Policy
      when ExcellentBandwidth
            then setLowLossRate
      when  HighBandwidth
            then setColorMode
            then setHighLossRate
      when MediumBandwidth
            then setGreyScaleMode
            then setMPEG2Decoder
      when lowBandwidth
            then setMPEG4Decoder
end
```

The previous example shows one adaptation policy that reacts to changes in the bandwidth. Four discrete ranges of bandwidth have been chosen to represent the space of bandwidth. For instance the threshold between the high and medium rate might be set to 100kbps and the threshold between low and medium rate might be set to 50kbps. In order to impact the frame rate, designers identified some actions, such as increasing the compression rate of the image encoding, decreasing the colour depth, or loading a different component that encapsulates a different compression (and decompression) algorithm.

In order to enable the execution of the self-adaptation policy, composite components need to communicate directly with their sub components. Each component is completed with a special port

which is devoted to communication with the composite component, that is to say the component which contains it. Figure 10 shows a part of the structure of the video player completed which those specific ports. Thus, a composite component can react to events or messages which come from probes but also from messages which come from functional components.

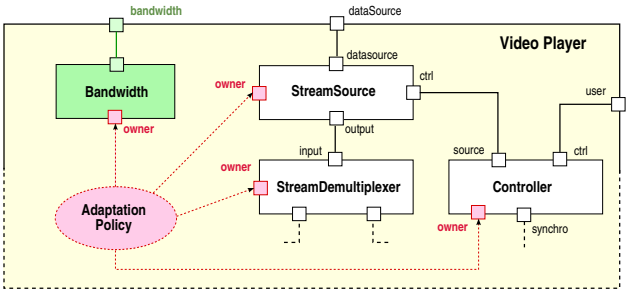

Figure 9 - The video player architecture with the "configuration port" which enable the communication between the composite component and its sub components

4. Early Validation using Simulation

The description of the component-based architecture used (see Section 3.1) is detailed enough to be simulated: the structure, the behaviour and the data of each component are described.

All the concepts which are involved in the description of software component have been described into a specific meta-model. This meta-model allows designers to describe the component architecture they want to build and to record it into a component model.

To simulate these architectural models, an operational semantic has been added to the meta-model. It enables the animation of any architectural model, since the behaviour of components is included in the architectural model. The Kermeta language [16] which enables the description of an operational semantic of a meta-model has been used to implement the meta-model and its semantic. However for the sake of conciseness this meta-model and it semantics are shown in this paper.

As explained in the section 3.1, the meta-model which as been used to capture component-based architecture share a lot of concept with the UML2.0 meta-model. The semantic which has been chosen for our meta-model freezes some ”semantic choices” for every semantic variation points concerned in the UML meta-model. For the sake of conciseness, all these variation points are not described in this paper, but the reader can refer to [5] to get more details about the management of semantic variation points.

The simulation of a component-based architecture which includes some adaptation policies might be done in 2 steps:

1. Model the self-adaptive component-based architecture and the adaptation policies which need to be validate.

2. Define an operational profile which will drive the simulation by defining the behaviour of environment of the architecture under test.

3. Simulate the architectural model using the simulator

4.1 Definition of operational profiles

An operational profile is a description of the environment around the architecture under test. For instance, it might be relevant to test the efficiency of the adaptation policies under an environment where the bandwidth is decreasing until it becomes very low. Such a scenario exercises adaptation policy and might happen if the user is moving far from the data source.

Operational profiles are described as stub components which close the architecture. An operational profile must provide a “test stub” port for each pending port in the component under test. Thus, the operational profile defines a test scenario which includes a sequence of calls to the features offered by the component under simulation. In the simulation we write to illustrate our simulator, every twenty simulation cycles, the operational profile decreases the bandwidth available for the video player component.

In this example we consider that the designer of our video player plans to deploy its video player on two kinds of platform: the first one enables high-level architectural adaptation actions (replacing the MPEG2 decoder by an MPEG4) whereas the second one is more basic and does not enables such actions. So the designer derives a second adaptation policy from the first one removing the actions which updated the architecture. He has to check to the performance of this new adaptation policy.

To sum up, the first adaptation policy decreases the loss rate, then switches to a grey scale mode and finally switches from an MPEG2 to an MPEG4 codec. The second adaptation policy is simpler and starts as well by decreasing the loss rate and then decreases the colour rate, but it does not switch from an MPEG2 codec to an MPEG4 codec. So the second adaptation policy does not include any architectural actions and can be deployed on a simpler execution platform than the first one.

4.2 Simulation results

Figures 11 and 12 show the evolution of the frame rate with respect to the bandwidth changes. The

frame rate that is measured throughout the simulation as the number of simulation cycles used to process a frame. In the following figures, we convert it as the number of frames processed by simulation step. Figure 11 presents the simulation obtained with a component that uses no adaptation policy. In this case, we use the BandwidthMonitor and the FrameRateMonitor components just for measuring the evolution of the frame rate. The frame rate is increasing as a function of the bandwidth.

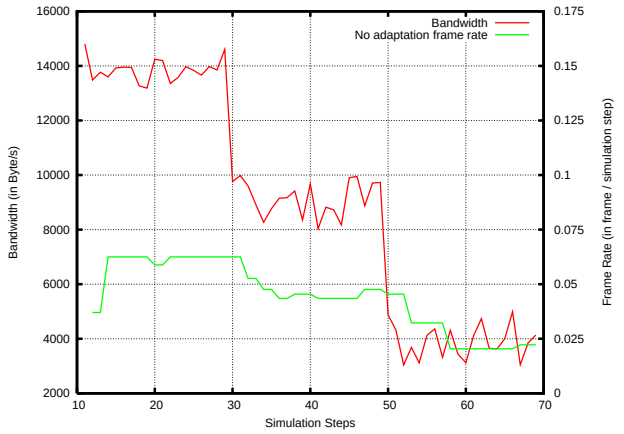

Figure 11 – The simulation results obtained without deploying any adaptation policy.

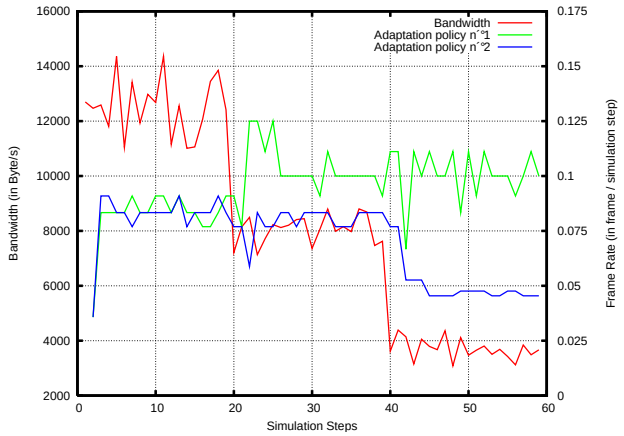

Figure 12 – Simulation results obtained with adaptation policies.

Figure 12 shows the result of the simulation of our two adaptation policies. In the figures 11 and 12, we can see the three levels of bandwidth which have been specified in the operational profile. As explained, in this simulation, the bandwidth is decreasing from a high level to a low one and simulation results show the performance of each adaptation policy (as a frame rate measurement).

The first adaptation policy maintains the frame rate when the bandwidth falls and even if the bandwidth becomes low.

The second adaptation policy keeps the frame rate when the bandwidth is ”medium” and just reduces the frame rate increase when the bandwidth is low. Since this adaptation policy does not use any architectural actions it can be deployed on a simpler (and cheaper) execution platform but the final product will not support a very low bandwidth.

However the results shown in this section are simple but illustrate our approach. In an industrial case-study, a set of simulations must be performed for each operational profile in order to get representative statistical data.

5. Code Generation for Fractal

The code generation has been developed as a two stages model transformation. First, the architecture description is mapped on a model of the platform such as a meta-model of the Fractal component model. Then, code generation is performed from this platform model to produce executable code.

Fractal is component based model which has been implemented in several ways. The main implementation is named Julia and is based on the Java language.

5.1 From architectural model to platform model

The transformation which produces a Fractal model from the description which has been introduced previously mainly concerns the structural aspects.

- The structural view is directly transformed into the structural concept of the Fractal platform. Each component is mapped onto a Fractal component and since Fractal is a hierarchical model, the hierarchy of component is maintained. Our model includes multiple ports on component (but not multiplicity on connector ends) and we use interface name to reify ports into the Fractal model.
- The description of the data handled by component cannot be used in the transformation. Indeed, that description is mainly included for the simulation purpose and it is an abstraction of the real data that will be processed by the system at run time.
- The behavioural view is currently not either used and the transformation which remains mainly focused on structural aspects. The Fractal model does not include any behavioural artefacts which would enable the reification of the behavioural description on the platform level.

5.2 From model platform to executable code

We implement a basic transformation which enables to Java code generation from the Fractal model. The challenge here is to map the component structure on the object-oriented model used in the Java implementation of Fractal. For instance, our high level description enables the description of component which implements a service in two

different ways (through two different ports, for instance).

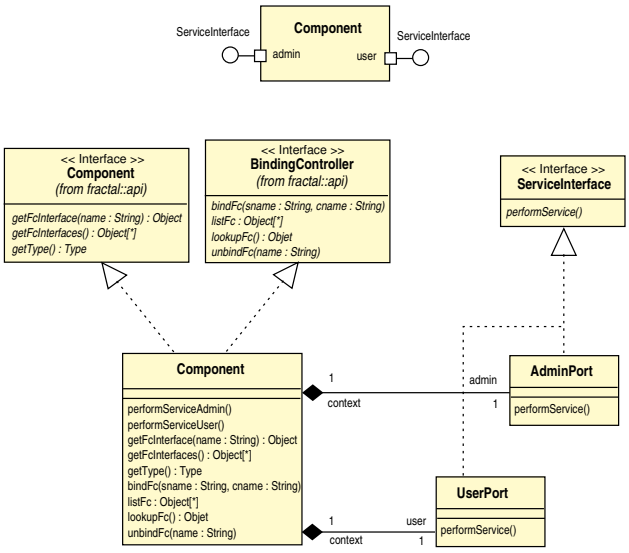

Figure 103 - A pattern used to implement multiple implementations of the same interface

Figure 13 shows the pattern which is applied to resolve multiple implementations of the same interface. A new class is created for each port of the component which implements the specific interface(s). These specific classes are connected to the component class which provides various implementations under different names (performServiceAdmin, performServiceUser). Then, the standard methods of the Component interface and of the BindingController interface must overridden to use the port objects.

Moreover, in order to make effective the adaptation policies, we have developed an extension of the Fractal platform which is able to directly execute rule-based adaptation policies as they are described into the section. Then, developers just need to complete the code skeletons of the functional components and of the probes to get the final executable artefacts.

6. Related Work

Balsamo and al. [2] survey several approaches to compute performance predictions on software models, and especially on UML models. Several approaches used queuing network-based methodologies; others use Stochastic Petri Nets, yet others are based on simulation. For the sake of conciseness we have selected here the most representative works on the area.

To describe extra-functional requirements, the OMG provides two specific UML profiles called SPT 1 and QoS. The SPT Profile [11] was a first attempt to extend UML with basic property definitions concerning time and performance. The QoS Profile [12], allows UML users to define a wide variety of extra-functional requirements. With these two profiles, one can annotate UML models with metrics that can be used to conduct performance analysis such as in [18]. Then these UML models are processed to obtain performance models (as Layered Queuing Networks) on which performance analysis can be conducted. Here, as the performance model is obtained by model transformations, there is no dynamic feedback possible between the performance model and the UML model, to take into account any architectural reconfigurations such as the one that is performed to change codecs in our example.

Another way to obtain performance analysis on UML models is described in [9]. As in previous approaches, the UML models are annotated using an ad-hoc profile, and then, transformed into a simulable model. Here, as the simulated model is the result of the Analysis Model Generator, the behaviour described in the original model cannot impact the quality attributes which are defined thanks to the profile. Thus, no feedback is possible between the quality information and the adaptation policies.

Based on ROOM and UML-RT, the work of V. Cortellessa [6, 7] enables the design of hardware resources in an object-oriented way. Then, on the functional side, the behaviours (expressed as statecharts) can be annotated with requests to the objects that represent the hardware resources. This approach enables some resource consumption prediction on ROOM models. Here we suggest to extend this approach to the Profile for Schedulability, Performance and Time express not only resources requests but also full component-based reconfiguration, including functional and extra-functional reconfiguration and to test them. Many Architecture Description Languages (ADLs) have been suggested in the literature [14], such as ACME [10], Darwin [13], ArchWare ADL [15], Wright [1], SOFA [17] and many others. All of them allow designers to address the structural features of component-oriented architectures and some of them have been extended to provide a way to describe the dynamics of the system such as Dynamic ACME, Dynamic Wright. However it is not possible in these approaches to manage component reconfiguration and to link it with functional behaviour.

Some works have already addressed the issue of specifying the dynamic reconfiguration of component-based architectures and many of them are surveyed into [4]. Among this, Allen and al. [1] present an extension of Wright which allows designers to deal with component-based reconfiguration but only on the functional side: nothing is suggested to deal with extra-functional properties. Batista and al. present in [3] an extension

of ACME to specify reconfiguration for a reflective component runtime called OpenCOM. There again,nothing is provided to link reconfiguration policies with QoS information at design time.

7. Conclusion

To design embedded systems, it is necessary to deal with high-level quality requirements which are often specified by the final user. Meeting these requirements has a lot of low-level quality implications such as memory or bandwidth consumption. Designers of such systems need to describe adaptation policies that maximise high-level quality properties with respect to changes in the low-level ones. However these adaptation policies are the result of various design choices that might have a deep impact on the final executable artefact and any rollback operation in such a design process has a very high cost. To avoid such problems, designers need to get support for the development of adaptation policies at design time.

The approach presented in this paper is based on new executable meta-modelling techniques which enable the creation of process devoted to support the development of self-adaptive systems. The use of a subset of the UML2.0 standard, makes possible to get early simulation results from component-based models which include adaptation policies.

To keep models executable throughout the design process, each quality property that is involved in adaptation policy is reified with a component which is able to monitor it. Then, the side effects of the functional behaviour of each component are described using service calls between functional and extra-functional components. With all the required elements reified in the design model, adaptation policies can be described as behaviours of composite components.

Executable models allow designers to perform simulation according to a specific operational profile that describes a test scenario. This approach enables to get suitable results to check if adaptation policies meet the high-level requirements or to select a particular adaptation policy among the set of possible ones.

Then a model transformation allows designers to get code skeletons where the adaptation policies have been fully generated but where the functional components and the probe components must be implemented.

8. References

[1] Robert Allen, Rémi Douence, and David Garlan: "*Specifying and analyzing dynamic software Architectures".* Lecture Notes in Computer qcience,1382:21–36, 1998.

[2] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. "*Model-based performance prediction in software development: A survey"*. IEEE Trans. Softw. Eng., 30(5):295–310, 2004.

[3] Thaıs Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. "*Managing dynamic reconfiguration in component-based systems*". In Ronald Morrison and Flavio Oquendo, editors, EWSA, volume 3527 of Lecture Notes in Computer Science, pages 1–17. Springer, 2005.

[4] Jeremy S. Bradbury, James R. Cordy, Jürgen Dingel, and Michel Wermelinger. "*A survey of self-management in dynamic software architecture specifications*". In David Garlan, Jeff Kramer, and Alexander L. Wolf, editors, WOSS, pages 28–33. ACM, 2004.

[5] Franck Chauvel and Jean-Marc Jézéquel. "*Code generation from UML models with semantic variation points".* In Lionel C. Briand and Clay Williams, editors, Proceedings of UML MoDELs 2005, Jamaica, volume 3713 of Lecture Notes in Computer Science, Montego Bay, Jamaica, October 2005. Springer.

[6] Vittorio Cortellessa and Maurizio Gentile. "*Performance modeling and validation of a software system in a RT-UML-based simulative environnement"*. In 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004), 12-14 May 2004, Vienna, Austria, pages 52–59. IEEE Computer Society, 2004.

[7] Vittorio Cortellessa, Pierluigi Pierini, and Daniele Rossi. "*On the adequacy of UML-RT for performance validation of an sdh telecommunication system"*. In Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), 18-20 May 2005, Seattle, WA, USA, pages 121–124. IEEE Computer Society, 2005.

[8] Nicola Cranley, Philip Perry, and Liam Murphy. "*User perception of adapting video quality"*. Int. J. Hum.-Comput. Stud., 64(8):637–647, 2006.

[9] Miguel de Miguel, Thomas Lambolais, Mehdi Hannouz, Stéphane Betgé-Brezetz, and Sophie Piekarec. "*UML extensions for the specification and evaluation of latency constraints in architectural models*". In WOSP '00: Proceedings of the 2nd international workshop on Software and performance, pages 83–88, New York, NY, USA, 2000. ACM Press.

[10] David Garlan, Robert T. Monroe, and David Wile. "*Acme: An architecture description interchange language"*. In J. Howard Johnson, editor, CASCON, page 7. IBM, 1997.

[11] Object Management Group. "*UML profile for schedulability, performance and time specification"*, September 2003. OMG Adopted Specification, formal/03-09-01.

[12] Object Management Group. "*UML profile for quality of service and fault tolerance characteristics and mechanisms"*, June 2004. OMG Adopted specification, formal/03-09-01.

[13] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. "*Specifying distributed software architectures".* In Wilhelm Sch¨afer and Pere

Botella, editors, ESEC, volume 989 of Lecture Notes in Computer Science, pages 137–153. Springer, 1995.

[14] Nenad Medvidovic and Richard N. Taylor. "*A classification and comparison framework for software architecture description languages".* IEEE Trans. Softw. Eng., 26(1):70–93, 2000.

[15] Ronald Morrison, Graham N. C. Kirby, Dharini Balasubramaniam, Kath Mickan, Flavio Oquendo, Sorana Cımpan, Brian Warboys, Bob Snowdon, and R. Mark Greenwood. "*Support for evolving software architectures in the archware ADL"*. In WICSA, pages 69–78. IEEE Computer Society, 2004.

[16] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. "*Weaving executability into object-oriented meta-languages"*. In Lionel C. Briand and Clay Williams, editors, Proceedings of UML MoDELs 2005, Jamaica, volume 3713 of Lecture Notes in Computer Science, Montego Bay, Jamaica, October 2005. Springer.

[17] Frantisek Plasil and Stanislav Visnovsky. "*Behavior protocols for software components"*. IEEE Trans. Softw. Eng., 28(11):1056–1076, 2002.

[18] Hui Shen and Dorina C. Petriu. "*Performance nalysis of UML models using aspect-oriented modelling techniques"*. In Lionel C. Briand and Clay Williams, editors, MoDELS, volume 3713 of Lecture Notes in Computer Science, pages 156–170. Springer, 2005.

[19] UML Revision Task Force. "*Unified Modelling Language 2.0: Superstructure specification"*. Official specification formal/05-07-04, Object Management Group, Needham, MA, USA, August 2005.