Mode-Based Reconfiguration of Critical Software Component Architectures

Etienne Borde [a, b] Grégory Haïk [a] Laurent Pautet [b]

[a] THALES Land and Joint Systems
5, avenue Carnot
91883 Massy cedex, France
firstname.lastname@thalesgroup.com

[b] TelecomParisTech
46, rue Barrault
75013 Paris, France
firstname.lastname@telecom-paristech.fr

***Abstract*—Designing reconfigurable yet critical embedded and complex systems (*i.e.* systems composed of different subsystems) requires making these systems adaptable while guaranteeing that they operate with respect to predefined safety properties. When it comes to complex systems, component-based software engineering methods provide solutions to master this complexity ("divide to conquer"). In addition, architecture description languages provide solutions to design and analyze critical and reconfigurable embedded systems. In this paper we propose a methodology that combines the benefits of these two approaches by leaning on both AADL and Lightweigth CCM standards. This methodology is materialized through a complete design process and an associated framework, MyCCM-HI, dedicated to designing reconfigurable, critical, and complex embedded systems.**[1]

I. Introduction

Embedded systems must automatically adapt to variations of their operational environment. In addition, these systems are often mission critical and must operate with respect to predefined safety properties through their utilization.

These two characteristics are opposed to one another insofar as the first one requires that the system be autonomous while the second one requires its execution be predictable. This problem has been partially tackled in AADL[2] [11] thanks to the notion of **operational modes**. AADL is an architecture description language – standardized by the SAE[3] – used to design and analyze software architectures of critical embedded systems. Additionally, our objective consists in providing a methodology that helps to tackle this issue all along the design process of complex systems (*i.e.* systems that are themselves composed of different subsystems). AADL lacks a *first class citizen* support for coarse grain software components. This abstraction level is yet a good intermeditate level between a system specification and a precise software architecture design. Consequently, AADL is not yet adapted for the design of complex systems.

In this paper, we propose a component-based methodology that improves the predictability of adaptable systems by modeling and restraining their reconfigurations. This methodology extends that of AADL for the management of operational modes to realm of component based software engineering and relies upon it. More precisely, it consists in representing (i) the different possible behaviors of a system thanks to operational modes, (ii) the conditions that limits and/or provoke the mode switches, and (iii) the impact of these mode switches on the software architecture described in terms of software components.

In addition to this process, we present a component framework (namely MyCCM-HI[4]) that automates the transformation of the component-based models into analyzable models, and that generates the corresponding technical code. This framework constitutes a major evolution of MyCCM [4] insofar as it reduces significantly the embeddability of the generated code, it supports the notion of operational modes and its impact on the software architecture, and respects the main implementation rules associated to critical software.

The remainder of this article is organized as follows : section II presents in more detail the problem we address in this paper. We present in section III, the different modeling artifacts on which our methodology relies. Section IV describes the design process associated with our methodology. In section V, we present the main results obtained with MyCCM-HI.Finally, section VI presents the conclusions of our work, and its future developments.

II. Motivation

We discuss in this section different interesting issues with regards to reconfigurable systems specification. First, it would be very useful to automatically discover the configuration that answers the best to the new operational environment of a system. The notion of contract, or the one of constraint resolution [5], have thus been proposed to answer this issue. The target configuration could then be computed by the system while it is running.

But in the case of mission-critical systems, we must guarantee during the design process that the system behavior will

[1]This work has been led in the scope of ANR/Flex-eWare and ITEA/SPICES projects

[2]Architecture Analysis and Design Language

[3]Society of Automotive Engineers

[4]Make your Component Container Model - High Integrity. MyCCM-HI is available at http://www.flex-eware.org under GPL since the end of January 2009.

978-3-9810801-5-5/DATE09 © 2009 EDAA

verify a predefined set of safety properties. This led us, as a first step, to consider reconfiguration specifications in which the source and target configurations are exhaustively defined at design-time. Indeed, this guarantees that the consistency of the target configuration can be verified at design-time, which is easier than ensuring the soundness of the decision algorithm that would determine at runtime the appropriate target configuration. Another issue consists in modeling formally the behavior of a system under reconfiguration, in order to ensure that the safety properties will also be verified during the reconfiguration process. Finally, the consideration of reconfiguration into an integrated design process would also be an important achievement in this domain.

In the academic community, research activities about software engineering dedicated to modes modeling and code generation are very few.

Actually, the community either focused on formal verification of systems under mode transitions [1], [3], [9], [10], or on the specification of conditions, evaluated at runtime, that can enable or disable reconfiguration (usually called "safe state") [7], [8], [13].

In industry, reconfiguration is key since the very first steps of a system engineering process consists in enumerating the different operational modes of the system and their associated functionalities. However, there is no automatic process for managing the impact of mode switches on the software architecture. This constitutes a gap between model-based system engineering and software engineering. Due to the lack of a dedicated modeling language and corresponding tools, this gap is still bridged manually. First, it makes the system difficult to design, update, and integrate since any modification of the specification of modes would require to modify the code in depth. Second, it makes difficult the formal verification of the system under reconfiguration since its behavior is hidden in the code.

Based on this observation, we have elaborated a component-based methodology that eases the analysis and the maintenance of reconfigurable and critical systems (i) by representing in the models the behavior of the system during a mode switch, (ii) by limiting the scope of this behavior to analyzable ones, and (iii) by generating most of the non-functional code of the software architecture (including the one that carries out this behavior).

The remainder of this article presents this methodology in deeper detail.

III. Modeling Artifacts

One of the objectives of our methodology consists in reducing the gap that exists between system engineering and software engineering due to the difficulty to associate an operational mode with a corresponding software architecture. To achieve this goal, we model a system, its subsystems and its mode automata. We then refine this system specification with a description of the software architecture and the impact of the mode switches on this software architecture.

In this section, we present the artifacts that compose these models.

A. Components Specification

A component is a piece of functionality that can be assembled with others in order to provide the full functional coverage of the system. Allowing to break down the whole system into smaller pieces, truly independently manageable, easier to develop and to reuse.

A component definition must describe the way it interacts with other components by means of ports. It then describes not only the services that the component is providing (as an object does), but also the ones it requires for functioning, to be provided by other components.

This allows the components to be connected externally from the application. Additionally, components describe their parameters, so that they can be configured externally.

MyCCM component definition is based on the OMG[5] Lightweight CCM [6] standard. Figure 1 illustrates the different kinds of ports a component may define: a facet (respectively receptacle) is a provided (*resp.* required) interface (*i.e.* set of operations), that implements synchronous communications, while events sinks and sources implement asynchronous communications. Finally, attributes are the parameters of the component for the configuration of which it provides accessors.

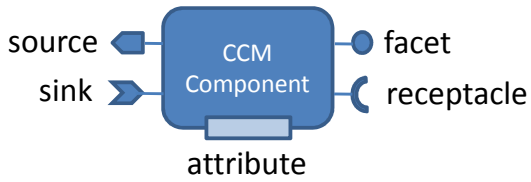

Fig. 1. Component Type

In our methodology, a software architecture is represented by a component that might be itself composed of subcomponents. A component composed of subcomponents is called a composite component whereas others are called primitive components.

A component definition consists in the specification a component type (its ports and attributes), one or more implementation of this type, and one or more instances of this implementation. The component implementation describes the content of the component, either as a set of subcomponents, or as a functional code provided by the component developer. Finally, the component instance enables to specify the target process, or the computational node (actually the address space), on which the component will be instantiated.

The presentation we made until now constitutes the background [4] of this paper. The remainder presents our contribution, that constitutes a major evolution of MyCCM component framework.

[5]Object Management Group

B. Modes Automata

In order to remain in the domain of verifiable systems, we restrain the reconfigurations to mode switches that are represented in MyCCM-HI models as communicating states automata: a mode corresponds to a state of the system in which it provides a set of functionalities corresponding to an expected behavior; an edge corresponds to a mode transition.

In MyCCM-HI, mode transitions mechanisms rely on :

- boolean expressions that enable this transition to be fired;
- the message value whose reception triggers the transition;
- actions that result from this mode transition.

In addition, these mode automata may interact with any other component in order to instantiate reconfiguration protocols that may depend on :

- the evolutions of the system operational conditions;
- the integrity of systems or subsystems that have to interact with each other;
- the propagation of mode switch in the different systems or subsystems that have to interact with each other.

This is the reason why mode switch mechanisms are described in the implementation of specific components called **mode automata** that are described with types, implementations and instances, and that can be connected with any other component.

We present hereafter the specificities of mode automaton components:

- their component type explicitly defines their modes;
- their ports are limited to event sources and sinks with a single data field defined as an enumerated type;
- their attributes are restricted to integers;
- their implementation describes the mode transition mechanisms in a specific language.

Listing 1. Operational Automaton

```
enum cmd {cmd_1, cmd_2};
eventtype m_cmd {
  public cmd a_cmd;
};
mode automaton M {
        mode M1(initial), M2;
  consumes m_cmd sink;
  publishes m_cmd source;
  attribute long attr ;
};
mode automaton implementation M_i implements M {
  in mode M1 :
  {
    [sink ?(a_cmd=cmd_2) AND (attr>2)] −−> M2
    {
      source !(a_cmd:=cmd_2);
      attr ++;
    };
  };
};
```

Listing 1 illustrates the specification of an operational automaton implementation *M_i*. In the definition of its type, *M*, we remark that its ports are event ports, and that their contents are limited to enumerated types. Besides, *M* defines two operational modes, *M1* and *M2*. In the definition of *M_i*, the mode transition between *M1* and *M2* is defined as follows: When the message value *cmd_2* is received on *sink*, if the value of the attribute *attr* is strictly superior to 2, then the new mode is *M2*, the value *cmd_2* is sent on *source* port and the value of *attr* is incremented.

By definition, a mode automaton is a subcomponent of a system or a composite, and defines the mode switch mechanisms of the corresponding system or subsystem.

In the following subsection, we present the characteristics of MyCCM-HI that enable to master precisely the real-time characteristics of the software architecture.

C. Real-time Software Architecture

As we said in section II of this paper, one of the benefits of our methodology is that it enables to generate most of the non-functional code of a system. This requires providing a precise description of the real-time software architecture. Here are the main elements of MyCCM-HI, inspired from AADL, that are dedicated to this description:

1) Activities: An activity is an active software entity likely to trigger a set of functional processing for which a deadline can be specified. There exists different kinds of activities, depending on the nature of the event that triggers it:

- Periodic activities, that are triggered by the real-time operating system clock each time a predefined amount of time has been spent;
- Aperiodic activities, triggered by the arrival of a software event;
- Sporadic activities, triggered by the arrival of a software event, but a minimal amount of time separating two executions of the activity;
- IT-based activities, triggered by the arrival of a hardware event.

The specification of the activity parameters depends on the activity type. For a periodic activity, we set the activity entry point – i.e. the first operation that will be called by the thread carrying out the activity – and the period of the activity. This operation actually references in the model an operation defined as component instance's facet. For sporadic activities, we specify the set of component instances' ports that will trigger an activity, and the period of this activity. Aperiodic activities are parametrized by the set of component instances' ports that will trigger the activity. Finally, IT based activities are parametrized with the IRQ that triggers the activity, and with the operation called when this interruption occurs. For schedulability analysis purposes, a deadline can be associated with an activity.

Listings 2 illustrates these specification rules: *C1_inst* corresponds to a primitive component instance, *f* is one of its facets, while *s1* and *s2* are parts of its event sinks. In this case, operation *f* of *C1_inst* is called every 10 milliseconds; the operation associated with *s1* (respectively *s2*) is called every

time a message is received on *s1* (*resp.* s2), if 100 milliseconds have past after the last trigger.

Listing 2. Activities

```
periodic activity A1
{
        configures C1_inst::f;
        period : 10 ms;
};
sporadic activity A2
{
        period : 100 ms;
        configures C1_inst::s1;
};
```

2) Execution Servers: When an activity is propagated across different processes, processors, or partitions, thanks to synchronous operation calls, one (or more) execution server is necessary to carry out the remote processing. This concept differs from the notion of activity since the execution server deadline depends on the activity deadline it interacts with, and should be deduced from it.

3) Shared data: Concurrency constitutes an error prone aspect of real-time systems programming, and an important aspect of schedulability analysis. In order to detect mistakes related to shared data usage, we propose to explicitly model shared data access into the component based architecture model.

D. Mode switches impact

Obviously, any modification of a system behavior does not require modifying its software architecture: it can be realized in an algorithmic way. However, three different reasons led us to model the impact of mode switches on the software architecture. First, the software architecture is sometimes impacted "de facto" (when a fault occurs); secondly, we decided to model the adaptation so as to ease the verification of the system behavior under reconfiguration and to generate the associated code; last but not least, we model the mode switches impacts, and generate the associated code, so as to ***limit the behavior of the system under reconfiguration to analyzable behaviors***. Indeed, the glue code generated to manage modes switches guarantees the consistency of data colocated with the mode automaton by using a synchronization mechanism between threads impacted by a mode switch and the code realizing this switch. Besides, this synchronization mechanism is based on analyzable patterns.

Let us go into a bit more detail this tricky problem. In a component-based approach, dynamic reconfiguration usually deals with components (i) connection/disconnection, (ii) instantiation/de-instantiation, and (iii) attribute value modifications. In addition, embedded systems must answer to requirements that may demand to activate/deactivate threads. These mechanisms impose to respect a reconfiguration protocol that belongs to the scope of analyzable reconfiguration policies, as described in [9].

In MyCCM-HI, an "**in mode**" clause is attached to components, activities, execution servers, connections, and attribute assignments in order to identify the set of modes in which those elements are valid. From this specification, MyCCM-HI computes the mode switch impact (*i.e.* the set of modifications of the architecture that have to be undertaken to reach the new configuration), and the set of impacted threads (*i.e.* the set of threads that have to finish their execution before a mode switch may occur). Both the code realizing the modes switch impact and its synchronization with the functional code is thus generated by MyCCM-HI.

In this section, we have presented the different artifacts of MyCCM-HI that helps to model complex, critical, and reconfigurable embedded systems. The following section describes how these elements are used in a complete design process.

IV. System-to-Software Design Process

The process we describe in this subsection is divided into two main steps that aim at modeling both the system and its software architecture.

A. System Level Specification

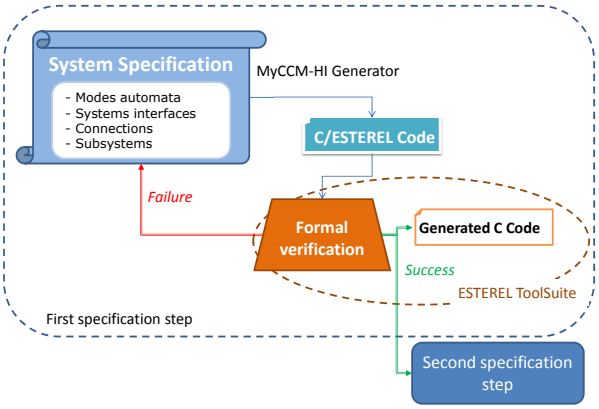

Fig. 2. MyCCM-HI Process: First Step

What we call a system level specification is exclusively composed of connected composite and operational automaton components. At this moment in the design process, and considering the amount of information we have at our disposal, many important properties can already be verified. For instance, we can verify that a system can alway be reinitialized, or that two subsystems will never be in conflicting modes: one in *manual* mode while the other is in *automatic* mode.

Figure 2 illustrates how the system level specification is used in a first design step: MyCCM-HI generators transform this specification into C code, or ESTEREL code so that we can use ESTEREL tool suite to generate certifiable C code and perform model checking and simulation.

Listing 3. Corresponding ESTEREL Code

```
data M_modes:
  type M_modes_type=enum{M1,M2};
end data
data m_cmd :
  type cmd_type=enum{mode_1,mode_2};
end data
interface M :
  extends data M_modes;
  extends data cmd;
  input sink: m_cmd;
  output source:m_cmd;
end interface
module M_i :
  extends M;
  var attr : integer :=0, current_mode: M_modes_type:=M1 in
    if :(current_mode=M1) then
      if :(? sink=mode_2 && attr>2) then
        emit ?source<=mode_2;
        current_mode:=M2;
      end if
    end if
  end var
end module
```

Listing 3 illustrates the result of the transformation of the operational automaton given in Listing 1: eventtype *m_cmd* is transformed into an ESTEREL data; operational automaton *M* is transformed into an ESTEREL interface, and the operational automaton implementation *M_i* is transformed into an ESTEREL module that contains the implementation of the mode transition specification.

Once their implementation code has been generated, operational automata are considered as primitive components, and consequently their integration in the software architecture is realized with the same techniques as for other primitive components.

In a second step of the process, the system level specification is refined into a software specification.

B. Software Level Specification

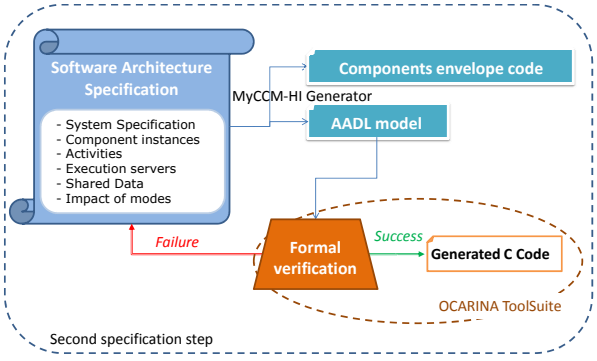

Fig. 3. MyCCM-HI Process: Second Step

What we call a software level specification refines the system specification. The refinement step consists in defining the primitive component instances, their processes, the activities, the execution servers, the shared data, and the impact of mode switches. From this specification, the user checks that the model conforms to the validation requirements (schedulability for instance), and uses MyCCM-HI generators to produce the code that implement the corresponding software architecture.

Figure 3 illustrates how the software level specification is used in this second design step: this specification is transformed into an AADL model. This model can be processed by the various tools analyzing AADL models like Cheddar [12] for schedulability analysis and Ocarina [14] for deadlock analysis. Last but not least Ocarina [14] generates from the AADL models, the code implementing connections, activities, execution severs, and shared data using sockets, threads, locks and other elements of the operating system API.

On Figure 3 we can see that the software specification is also used to generate the components' envelope code. This code is used for primitive components initialization (attribute values definition and connections), for the corresponding reconfigurations (modification of attribute values, and disconnection/reconnection), and for interfacing the code generated by Ocarina with the user code contained in the primitive components implementation. In addition to the verifications presented until now, we propose to apply schedulability analysis to the code implementing the mode switches: to do so, we would need to include in the generated AADL model the description of the mode switches specified in MyCCM-HI and use the AADL mode analyzers such as [3], [10].

Figure 4 sums up the final architecture of an executable code obtained with MyCCM-HI and Ocarina: AADL runtime code references the code generated by Ocarina; the components' envelope code is the interface code generated by MyCCM-HI; finally, the primitive components implementation either corresponds to functional code provided by the component developer, or corresponds to the mode automata implementation code generated by MyCCM-HI (see paragraph IV-A).

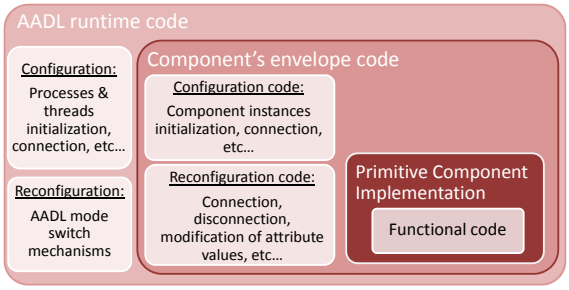

Fig. 4. Resulting Software Architecture

We present in next section the main results obtained with MyCCM-HI on a little example.

V. Results

In order to illustrate the capacity of our approach to address real-time and embedded systems domain, we have modeled a very simple example, from which the whole runtime code has

been generated. This example is composed of two processes : one client and one server.

The client process is composed of two threads (one sporadic and one periodic), and the server process is composed of one sporadic thread.

Two components are instantiated in the client process, and one in the server process. The corresponding footprint measures give a minimal size of 24.4 kB for the client process, and 20.9 kB for the server process. This results shows that we have a very important improvement in the embeddability of the generated processes.

Besides, we have built examples that illustrate the usage of operational modes in order to modify dynamically the connections and attribute values of components of the software architecture.

Last but not least, the generated code respects the following programming patterns so as to make it analyzable: no *pointer-to-function*, no dynamic memory allocation (including reconfiguration mechanisms), no unbound loop, and no unbound data types.

Thus, MyCCM-HI briges the gap between system and software engineering in the domain of real-time and embedded systems while adressing specific requirements of critical systems.

VI. Concluding Remarks and Future Works

The contribution presented in this paper consists in a methodology that reduces the gap between the model-based system and software engineering by automating the production of the code implementing the impact of operational modes on the software architecture. Besides, this process, and the associated tools permit to simulate and verify properties relative to this specific code.

The perspective of the research activity described in this paper is the the integration of error models into the system level description. Indeed, errors are often cited as being causes of mode switch. In particular, this is the way we intend to capture the specification of the expected behavior of the system upon detection of a faulty hardware.

References

[1] L. Apvrille, P. de Saqui-Sannes, P. Sénac, and C. Lohr. Verifying service continuity in a dynamic reconfiguration procedure : Application to a satellite system. *Software Engineering, Vol.11, No.2*, April 2004.

[2] H. Balp, E. Borde, G. Haïk, and J.-F. Tilman. Automatic composition of aadl models for the verification of critical component-based embedded systems. *13th IEEE International Conference on Engineering of Complex Computer Systems*, April 2008.

[3] D. Bertrand, A.-M. Déplanche, S. Faucou, and O.-H. Roux. A study of the aadl mode change protocol. *13th IEEE International Conference on Engineering of Complex Computer Systems*, April 2008.

[4] E. Borde, G. Haïk, V. Watine, and L. Pautet. Really hard time developing hard real time. *National Workshop on Control Architectures of Robots*, 2007.

[5] G. Grondin, N. Bouraqadi, and L. Vercouter. Assemblage automatique de composants pour la construction dagents avec madcar. *2e Journe Multi-Agent et Composant (JMAC06)*, March 2006.

[6] O. M. Group. Light weight corba component model revised submission. *OMG Document realtime/03-05-05 edn*, May 2003.

[7] J. Kramer and J. Magee. The evolving philosophers' problem: Dynamic change management. *IEEE Transactions on Software Engineering*, November 1990.

[8] J. Polakovic, A.-E. Ozcan, and J.-B. Stefani. Building reconfigurable component-based os with think. *Software Engineering and Advanced Applications. 32nd EUROMICRO Conference*, August 2006.

[9] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new protocol. *Real-time systems 26(2)*, 2004.

[10] J.-F. Rolland, J.-P. Bodeveix, M. Filali, D. Chemouil, and D. Thomas. Modes in asynchronous systems. *13th IEEE International Conference on Engineering of Complex Computer Systems*, April 2008.

[11] SAE. Architecture analysis and design language. *Technical Report SAE AS5506, SAE*, 2004.

[12] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with aadl. *Proceedings of the 2005 annual ACM SIGAda international conference on Ada*, November 2005.

[13] U.Brinkschulte, E. Schneider, and F. Picioroaga. Dynamic real-time reconfiguration in distributed systems: Timming issues and solutions. *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 05)*, May 2005.

[14] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. *11th IEEE International Conference on Object-oriented Real-time distributed Computing (ISORC'08)*, May 2008.