This is an author-deposited version published in: http://oatao.univ-toulouse.fr/
Eprints ID: 3663

To cite this document : DELANGE, Julien and HUGUES, Jérôme and PAUTET, Laurent and DE NIZ, Diosisio. A MDE-based Process for the Design, Implementation and Validation of Safety Critical Systems. In : *Proceedings of the 5th UML AADL Workshop (UML AADL 2010), Oxford, 24 March 2010*

Any correspondence concerning this service should be sent to the repository administrator: staff-oatao@inp-toulouse.fr

An MDE-based Process for the Design, Implementation and Validation of Safety-Critical Systems

Julien Delange, Laurent Pautet
TELECOM ParisTech - LTCI
46, rue Barrault
F-75634 Paris CEDEX 13
{delange,pautet}@enst.fr

Jérôme Hugues
Université de Toulouse, ISAE
10, avenue E. Belin - BP 54032,
F-31055 Toulouse CEDEX 4
jerome.hugues@isae.fr

Dionisio de Niz
SEI/CMU
4500 Fifth Avenue
Pittsburgh, PA, USA
dionisio@sei.cmu.edu

Abstract

Distributed Real-Time Embedded (DRE) systems have critical requirements that need to be verified. They are either related to functional (e.g. stability of a furnace controller) or non-functional (e.g. meeting deadlines) aspects.

Model-Driven Engineering (MDE) tools have emerged to ease DRE systems design. These tools are also capable of generating code. However, these tools either focus on the functional aspects or on the runtime architecture. Hence, the development cycle is partitioned into pieces with heterogeneous modeling notations and poor coordination.

In this paper, we propose a MDE-based process to create DRE systems without manual coding. We show how to integrate functional and architecture concerns in a unified process. We use industry-proven modeling languages to design functional elements of the system, and automatically integrate them using our AADL toolchain.

1 Introduction

The design of a DRE system involves multiple tightly integrated domains. Cooperation from these domains is essential for the successful development of these systems.

Model-Driven approaches [5, 9] seeks a solution to the integration problem: all is model, representing either problem or solution artifacts. These approaches improve software development and application reliability.

Two MDE development approaches had emerged for DRE systems. The first one focuses on functional parts [2, 3] (application specific code) while the other details the overall architecture [4, 1]. In this paper, we propose an approach to integrate functional aspects with architecture approaches and develop the whole system from models.

DRE systems are composed of functional (e.g. control algorithms) and non-functional (e.g. threads) aspects that are loosely coupled. An ideal development process would integrate them to take benefits from both methods.

Architecture-driven approaches can partly generate implementation code from high-level designs, yet the integration of functional code in an architectural skeleton remains hand-made. For instance, Simulink or SCADE models cannot be integrated automatically with RTOS drivers nor deploy over a distributed system with specific hardware.

In this paper we describe an approach (see figure 1) to integrate architecture and application models. It relies on two modeling languages: an application-level language (Simulink or SCADE) and an architecture-level language (AADL [8]). Application code is generated using vendor-specific tools (step 2) and architecture-level code that uses application-level code is also generated (step 3). Merging both approaches (step 4) produces the final implementation.

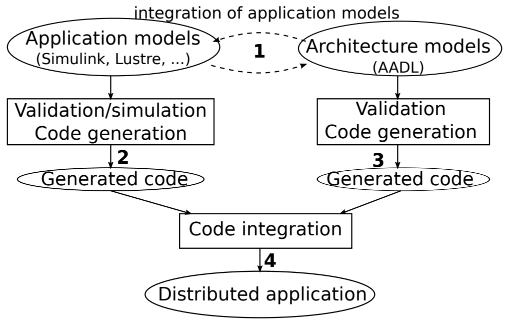

Figure 1. Proposed development process

2 Architecture Modeling with AADL

AADL is a component-centric language which focuses on the description of the non-functional aspects of the components such as timing or memory concerns.

An AADL description is made out of hardware and software *components*. The AADL standard defines basic component types that must be extended to describe a system. The software types are:`data`, `thread`, `thread group`, `subprogram`, `process`. The execution platform components types are `memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`. Finally, there is one hybrid component type (`system`).

Components describe well identified elements of the actual architecture. The *Subprogram* type models application code and references another external (application) model. The *Thread* type models the active part of an application (such as POSIX threads). The *Process* type models an address space that contains the *threads*. The *Processor* type models aspects of both the processor and the operating system relevant to the non-functional properties. The *memory* type models hard disks, RAMs, ROMs and other forms of memory. The *bus* type models all kinds of networks and hardware connections. Finally, the *system* type models composite components that are made up of hardware and software components. For example, a *system* may represent a board with multiple processors and memory chips.

Components are organized in a hierarchy, i.e.: components contain other components (called subcomponents in this case). AADL models contain a topmost system component that contains, hardware and software subcomponents and the deployment of the software to the hardware.

The interface specification of a component is called its *type*. An interface is a collection of *features* that can be connected to other component's features to model their communication. Each component has a separate *implementation* description that is populated with subcomponents and connections among them. An implementation of a thread or a subprogram can specify *call sequences* to other subprograms, thus describing the execution flows in the architecture.

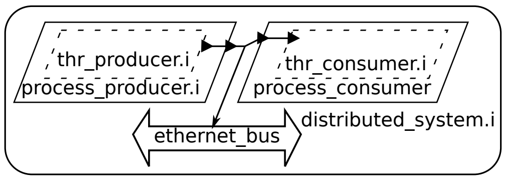

Figure 2. Sample AADL model

AADL associates *properties* to model elements. Properties are name/value pairs that represent components characteristics and constraints, such a the period of a thread, etc.

Figure 2 shows an AADL model using its graphical representation. In this model, two processes communicate through the network. Each `process` contains a `thread`, one being a producer, the other a consumer.

Interested readers can find an introduction to this language in [7].

3 Modeling functional blocks

3.1 Simulink

Simulink is one of the modeling languages of Matlab, a simulation tool commonly used in the industry. One of the key reasons for the popularity of Simulink is the mathematical toolboxes that enable the designer to simulate key characteristics of the system.

Simulink is a dataflow language. A Simulink system is composed of data-processing blocks and connections between their inputs and outputs A typical model in Simulink includes both the system under control, and its associated controller. As a result, both parts of the model can be exercised by the simulation engine, to focus on the evaluation of the mathematical functions.

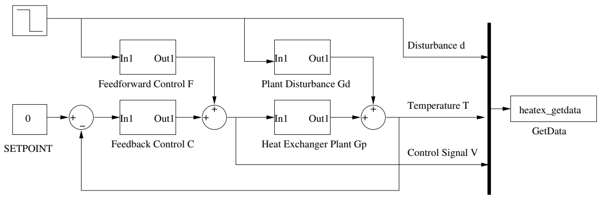

Figure 3. Simulink Example - Heat Exchanger

As an example, consider the heat exchanger system depicted in Figure 3. In this model both the controller modules (`Feedback Control C` and `Feedfordward Control F`) and the controlled system (`Heat Exchange Plant Gp`) are present. In this example, the steps of both, the computations of the controller modules and the reactions of the controlled system simulations, are advanced synchronously by the simulation engine.

However, the final generated code to be used in the real system has not controlled plant code but another piece of software of hardware. Hence, we need to adapt the controller algorithm to synchronize with the plant. On the one hand, it needs to run periodically at a rate that can keep up with the dynamism of the plant. On the other hand, feedback loops need to be implemented to communicate its value from one activation of the controller to the next.

Both, the periodicity of the controller, and the feedback connections are part of an architectural model that complements this functional model. On the one hand, the periodicity is modeled as periodic threads that call the appropriate functional code. On the other hand, the feedback loops are modeled as delayed connections that ensure consistency in the communication between the threads. All these elements are part of the concerns to be described in an AADL model.

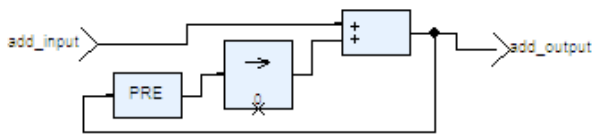

Figure 4. SCADE code example

3.2 SCADE

“Safety Critical Application Development Environment” (SCADE) defines a simple and efficient textual formalism to design safety-critical software. SCADE relies on a graphical notation to model data flows. The design of SCADE programs is made out of computing blocks with data flows that connect blocks inputs and outputs (see figure 4 for an example). State machines can also be used to control program behavior.

Applications written with SCADE can be simulated, which help the developers to test and ensure their correctness. SCADE can automatically generates code from the graphical notation using its code generator (`kcg`). This code generator outputs potentially certified C code. Such a development process help the developer to design its application and reduce certification costs. SCADE was successfully used in the industry, especially in the avionics domain.

Figure 4 depicts a SCADE sample program that adds the value of its input to a global memory. The input `add_input` is added to the previous computed value. Then, the output `add_output` contains the computed value.

However, code generated by `kcg` still requires manual integration with other elements (such as drivers) as well as a separate toolchain for integration on target hosts. Deployment/integration of verified applications in validated architectures would make the development process more robust.

4 Modeling process

4.1 Application-driven process

In this approach, the functional part of the system is first designed and then, allocated to a runtime architecture. Techniques to do this include the use of bin-packing algorithms to allocate software to hardware, design exploration to find the best assignments, and analysis to ensure the system is schedulable. Each of these steps involves:

1. Functional blocks are defined using vendor-specific tools. These tools allow the simulation and analysis of these blocks to verify their correctness and properties.

2. An AADL runtime architecture is designed, listing `processor`, `memory` and `bus` components. This architecture defines hardware elements and their topology.

3. Functional blocks are imported as AADL `subprograms` and `data` components.

4. Finally, the allocation is performed. It takes advantage of the annotations in the AADL model that includes WCET, bandwidth/memory consumption, etc.

This development process was used in our Simulink case-study, detailed in section 6.1

4.2 Architecture-driven process

In this approach, the architecture is first designed describing the runtime components of the system, e.g. processes, threads, processors, etc. Then, functional blocks are then allocated to these components. This process is based on these following steps:

1. An AADL architecture is designed and list available resources (including `threads` with scheduling concerns, root `subprograms` and `data` types).

2. Functional blocks are implemented. Their interface should match the AADL interface.

3. AADL `subprograms` and `data` components are then refined to reference these implementations.

Such a development process was used in our SCADE case-study, detailed in section 6.2.

4.3 Integration challenges

The integration of the functional model with the runtime architecture model allows us to validate and generate the complete system.Using MDE principles (abstraction, model transformation and analysis), this creates a complete path from models to code, with integrated validation and simulation.

However, the model-to-code process that integrates architectural and functional model must ensure the consistency of such integration. There are three types of consistency:

• **Data representation consistency**: Data must have the same representation in architecture and application models.

• **Source-code interoperability**: Application models use a specific API to exchange data between blocks. This API must be integrated in the architecture code to import/-export data between architecture and application code.

• **Semantics consistency**: The integration process must preserve models semantics and data flows.

5 Supporting the process

5.1 Import application data in AADL

In Simulink or SCADE, the designer specifies the data types used on each component. This information should be used to pre-allocate buffers to exchange data in the runtime system. For this reason, it should be integreated in AADL models so that data components of the architecture-level models reflect types used in application level models.

In AADL, data are described using the `data` component and their characteristics are described by adding `properties` to it. So, AADL `data` components must ensure the **data consistency** between the application and architecture models.

For example, the modeling of an SCADE integer in AADL is achieved with a `data` component associated with an appropriate property (see listing 1).

```
data integer
properties
   Data_Representation => Integer;
   Source_Language => Scade;
end integer;
```

Listing 1. Map SCADE integer using AADL

5.2 Import application functions

Traditional code (written in Ada or C) integration in AADL models is achieved with the `subprogram` component. It models an instruction flow executed by a `thread`. Application-level non-functional properties are described by adding properties to AADL components. These properties are reused in the process development for compilation or integration purposes. To integrate application-level models into AADL models, we specify modeling patterns that rely on specific properties. The mapping of SCADE or Simulink application code is described below.

5.2.1 Simulink mapping

Integrating Simulink models in AADL is achieved by adding some properties to a `subprogram` component:

- `Source_Language` is set to *Simulink* .
- `Source_Name` specifies the name of the Simulink block.
- The `Source_Text` property specifies the directory that contains the code generated by the Simulink code generator.

Finally, we define the association between `subprogram features` and the Simulink block inputs/outputs so that the architecture model know its mapping with Simulink communication ports.

Listing 2 corresponds to the integration of the `Heat Exchanger` block (shown in figure 3). We map this application in an AADL subprogram and specify the name of the mapped Simulink block using the `Source_Name` property. Then, we map Simulink output signals to AADL subprogram features using the `Source_Name` property on each feature. Here, the AADL `dist` feature corresponds to the `disturbance` Simulink signal, the AADL `temp` feature corresponds to the `temperature` Simulink signal and the AADL `csignal` feature corresponds to the `Control Signal` Simulink signal.

```
subprogram spg_simulink
features
        dist : out parameter simulink_real
            {Source_Name => "disturbance";};
        temp : out parameter simulink_real
            {Source_Name => "temperature";};
        csignal : out parameter simulink_real
            {Source_Name => "Control Signal";};
properties
        source_name => "Heater Exchanger";
        source_language => Simulink;
        source_location => "/path/to/simulink-code/";
end spg_simulink;
```

Listing 2. Simulink's `Heat exchanger` in AADL

5.2.2 SCADE mapping

Importing a SCADE node with AADL is achieved using an `subprogram` component with the following properties:

- `Source_Language` is set to *Scade*.
- `Source_Name` specifies the name of the SCADE node.
- `Source_Location` points the location of the SCADE code on the filesystem.

We also indicate the mapping of AADL features onto SCADE parameter by adding the `Source_Name` property on each feature. The value of this property is set the their corresponding inputs/outputs in the SCADE model.

The listing 3 shows the integration of the SCADE adder example (shown in figure 4). The property `Source_Name` is added on each parameter to describe the mapping between the AADL `subprogram` features and SCADE inputs/outputs. Here, the `input` parameter from the AADL component corresponds to the `add_input` input of the SCADE node and the `output` parameter corresponds to the `add_output` output of the SCADE model.

```
subprogram spg_scade
features
input: in parameter integer {Source_Name => "add_input";};
output: out parameter integer {Source_Name => "add_output";};
properties
        source_name => "inc";
        source_language => Scade;
        source_location => "/path/to/scade-code/";
end spg_scade;
```

Listing 3. `inc` SCADE node with AADL

5.3 Code generation and integration

This section details the characteristics of architecture and application generated code and explains their integration.

5.3.1 Architecture-level code overview

Architecture-level generated code provides resources to execute functional code and can be seen in three parts:

• **Thread part** reflects AADL `thread` components. This part performs calls to the application-level code, as described in the `subprogram` components.

• **Process part** creates resources (`threads` or `shared data`) and establishes communication channels with the other nodes of the distributed system.

• **OS and network configuration** part is generated using information from the `processor` and `system` components. It configures the underlying operating systems of each node to execute the generated code (for example, RTEMS [6] needs declarations to enable specific functionalities).

5.3.2 Application-level code overview

Application-level code generators (as in Simulink or SCADE) provides two important functionalities:

• An **initialization function** that instantiates resources.

• A **reaction function** that computes outputs from inputs values. Calling this function is the execution of one *step* in the computation of the application model.

These code generators provide mechanisms to interoperate with the generated functions (get/put data into the application code, call the reaction function, ...). In our case, we use these features to exchange data between application and architecture levels and trigger application reaction.

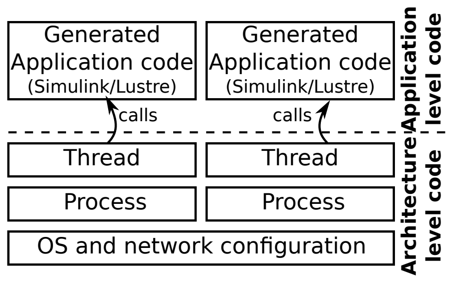
Generated Application code (Simulink/Lustre)
Generated Application code (Simulink/Lustre)
calls
calls
Thread
Thread
Process
Process
OS and network configuration
Application level code
Architecture level code

Figure 5. Interaction between application and architecture generated code

5.3.3 Integration of the code from both generators

Application level code is described using AADL `subprogram` components so that there is no difference between traditional subprogram and application model subprograms: application code resides on top of the thread part of the architecture code, as shown in figure 5. Architecture code remains unchanged. Changes are only made in the generated `subprogram`: this part enables interactions (data exchange) with application code and calls the reaction function.

To do so, the code generator creates a function from the AADL subprogram that follows this pattern:

1 It injects the `in` parameters of the AADL `subprogram` into the application.

2 It calls the **reaction function** of the application-level code to compute new output values.

3 It retrieves application-level outputs injects them into the AADL subprogram output parameters

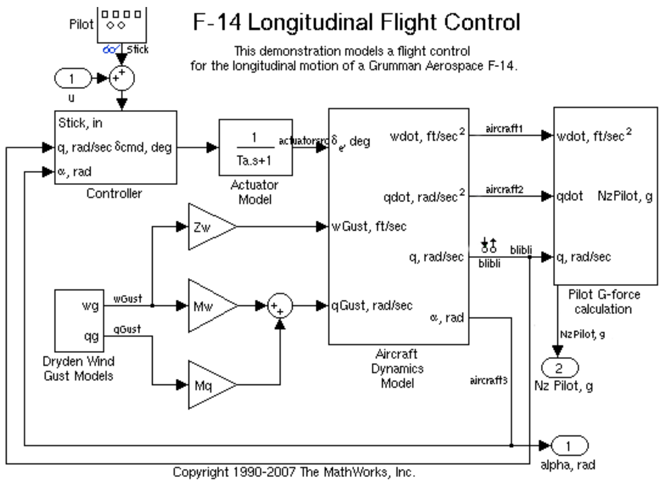

Figure 6. Simulink model of our case-study

6 Case studies

During our tests, application code was generated with dedicated tools (Real-Time Workshop for Simulink, KCG for SCADE) and architecture code was generated using the OCARINA tool suite. It generates C code with the appropriate glue code detailed in section 5.

6.1 Simulink case-study

This case-study follows an application-driven process: a flight control application (illustrated in figure 6 and available in Simulink releases) was divided in three parts (*controller*, *aircraft control* and *pilot calculation*) and deployed on a distributed architecture (illustrated in 7).

We take into account the requirements of the application (feedback loops, data flows, ...) to preserve application requirements in the architecture.

6.2 SCADE case study

The SCADE case study follows an architecture-driven process: we first designed a distributed architecture (shown

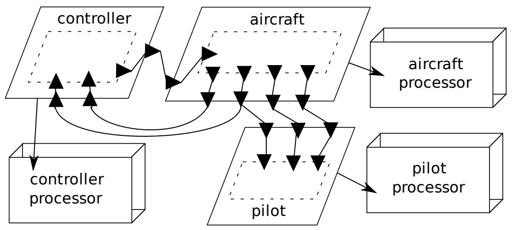

Figure 7. Simulink case-study architecture

in figure 8) with the appropriate requirements and integrate an application model (a cruise control system and available in SCADE releases). Application inputs are simulated by a separate node (`src_process`).

In the architecture side, one node (`src_process`) simulates application values and send them to another node through the network. This node (`scade_process`) receives the values, computes a new value and sends it to a third node. It `dst_process`) executes a subprogram that prints the speed value on the screen.

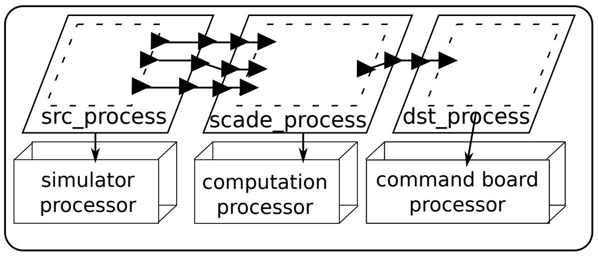

Figure 8. SCADE case-study architecture

6.3 Integration validation

We validated our approach by comparing simulation and execution. The simulated values were obtained with dedicated tools (Matlab or SCADE toolsuite). To collect values during execution we instrumented the code to get values sent/received over `data ports`. Then, we compare simulation and execution values according to the mapping rules between application and architecture models.

During our experiments, we found that values from simulation and execution were the same. It shows the correctness of our approach and demonstrates that application and architecture code are correctly integrated.

7 Conclusion

In this article, we presented a full MDE approach to design and implement DRE systems. We detailed different modeling approaches (architecture- or application-driven) and explained how architecture and application models are integrated from design to implementation. Code was automatically generated from integrated models to create distributed applications without manual coding. This development process avoids error from traditional coding methods and preserves the analysis results from the modeling tools.

An additional advantage of using AADL is the continuous flow of analysis tools that are based on this language. That is, both the research community and the industry are using AADL as a platform to explore new analysis techniques. The use of these analysis tools, combined with an appropriate development process (such as the one detailed in this paper) would ease DRE systems development.

References

[1] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation and control. 6(2):201–227, June 1996.

[2] K. Hales. Matlab/simulink model as a tool for process design and commissioning. May 2004.

[3] R. H. Martin and D. Raffo. A model of the software development process using both continuous and discrete models. *Software Process: Improvement and Practice*, 5(2-3):147–157, 2000.

[4] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000.

[5] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[6] OARCorp. Rtems - http://www.rtems.com.

[7] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis and Design Language (AADL) : An introduction. Technical report, 02 2006.

[8] SAE. *Architecture Analysis & Design Language v2.0 (AS5506)*, September 2008.

[9] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.