Code generation from AADL to a real-time operating system: an experimentation feedback on the use of model transformation

Matthias Brun  
ESEO  
TRAME research team  
49000 Angers, France  
matthias.brun@eseo.fr

Jérôme Delatour  
ESEO  
TRAME research team  
49000 Angers, France  
jerome.delatour@eseo.fr

Yvon Trinquet  
IRCCyN (UMR CNRS 6597)  
Real-Time System Team  
44000 Nantes, France  
yvon.trinquet@irccyn.ec-nantes.fr

Abstract

Several approaches, such as the UML MARTE profile or AADL start to reach maturity for the design of Real-Time Embedded System (RTES). The use of such formalisms and their associated verification tools relies on the confidence of the designer in the successful translation of these high-level descriptions into correct executable code. Part of this translation is performed by code generators. However, code generators are often black boxes or difficult to customize. This fact conflicts with the specific needs of the development of RTES where different code generation strategies could be involved.

Recently, Model Driven Architecture (MDA) has offered sophisticated tools for model transformation. This paper presents an experimentation: code generation from an AADL model to C code using MDA tools. Based on this experimentation, statements on the interest of MDA tools for this purpose are given. Beyond this feedback, a set of open questions emerged about the need of flexibility of code generators and the different ways for setting this flexibility in MDA tools.

1. Introduction

During the last decade, several approaches, such as UML profiles (STP and its incoming successor MARTE) or AADL, have gained maturity for the design of Real-Time Embedded System (RTES). Their ability to express real-time features has improved; additional verification and validation tools appeared and can now be used to assist designers with overcoming the increasing complexity of real-time systems.

The use of such formalisms and their associated verification tools relies on the confidence of the designer in the successful translation of high-level descriptions into correct executable code. Code generators have been developed in order to automate part of this translation work. Their use limits the possibility of introducing errors, reduces the time required for system development, prevents designers from making fastidious repetitive basic code translations, etc.

However, the use of code generators in RTES development faces several problems. Beyond the difficulties of their development, a recurrent reproach is that code generators are often black boxes or difficult to customize. This fact conflicts with the specific needs of the development of RTES: different code generation strategies could be involved depending on the conflicting implementation requirements of the RTES (e.g., performance vs reliability), the domain (for instance, avionic code standards differ from spatial code standards...), business rules, etc. Moreover, this opacity of code generators has another drawback: designers are reluctant to trust them (due to their ignorance of the code generation strategy used) and therefore avoid to use them; validation and verification of code generators are more difficult to achieve.

Recently, Model Driven Architecture (MDA) [5] has offers sophisticated tools for model transformation. These tools seem interesting for developing adaptable code generators. This paper presents feedback (benefits and also lacks) deduced from one specific experimentation: code generation from a sub-part of an AADL (international standard SAE) [18] description to C code compliant with the OSEK/VDX real-time operating system. Beyond this feedback, a set of open questions emerged about the need of flexibility in code generation.

This paper is divided into three parts. The first part describes the experimentation, its purpose and results. The second part presents our feedback. The last part summarizes our position and future works.

1 The described work is part of the TOPCASED project.
2 Experimentation: From AADL to OSEK/VDX-compliant code

2.1 Context of the experimentation

MDA provides an approach to model-based software development, where models are representations or views of a system under study (e.g., a UML description or a piece of code). A model should be conform to a metamodel. A metamodel is a precise specification of the considered concerns the model will represent. Based on these metamodels and using transformation languages, one can express the generation of a set of target models from a set of source models. Two kinds of transformation can be considered: «model to model» transformation, where target and source models must conform to metamodels and «model to text» transformation, where only source models are conform to metamodels.

Code generation from a model can be seen as a «model to text» translation, possibly using intermediate «model to model» steps. One of these steps may be the generation of an «abstract syntax tree» representation of the textual code to generate.

Using «model to model» before «model to text» allows incremental refinement. For instance, we can first generate a procedural code model and then translate this model into C or Ada textual code. The intermediate models play the role of a «pivot» from which several implementation alternatives may follow. This last technique is used in compilers (with a front-end and a back-end language).

As stated previously, our research team currently investigates these different approaches. In this paper, we present an experimentation where transformations directly generate code models expressed in specific languages (OIL and C) and compliant with a specific standard (OSEK/VDX). In order to carry the experimentation, we used the AMMA (Atlas Model Management Architecture) [3] platform. It provides MDA tools which allow to define metamodels, model transformations (with Atlas Transformation Language - ATL [2]) and ways to generate parsers and pretty-printers (the latter could be seen as «model to text» transformations).

2.2 AADL and OSEK/VDX

In recent trends, AADL (international standard SAE) [18] has proved a good candidate as a modeling language for real-time embedded systems. AADL defines component classifiers and connectors specific to the RTES domain. Component classifiers are either software components (process, thread, subprogram, data, ...), execution platform components (processor, memory, bus, device, ...), or composite components. In a complete description, the software components are bound to execution platform components required for executing them. An architecture model of the operational system can then be instantiated.

AADL was used originally to specify avionic systems, but nowadays intends to describe a wider range of architectures for performance-critical, real-time embedded systems. From this perspective, that experimentation considers AADL to design vehicle-domain systems where OSEK/VDX (Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive) [6] is a representative industry standard.

Various parts of OSEK/VDX specify: the basic services of the real-time kernel (OS), the communication services (COM), the network management services (NM) and a configuration language (OIL - OSEK Implementation Language). In this work, we only consider centralized architectures (OS and OIL specifications). OSEK/VDX OS [16] is a static kernel (no dynamic object creation) which mainly provides services to handle tasks, shared resources, events and alarms. OSEK/VDX OIL [15] is used to describe objects (tasks, resources, alarms, etc.) in order to generate kernel configuration files to build a binary image of the application.

2.3 Experimentation

In the experimentation, two transformations generate either C language code (compliant with OSEK/VDX) and corresponding OIL configuration code. Figure 1 illustrates the performed model transformations.

The AADL metamodel and models used come from OSAE (Open Source AADL Tool Environment) [19], a set of

\[\text{Figure 1. Model transformations from AADL to OIL and C compliant with OSEK/VDX}\]

\[\text{2 these kinds of components symbolize both hardware and associated softwares or protocols.}\]
AADL tools based on Eclipse [1]. The AADL metamodel is divided into seven packages: core, component, feature, connection, flow, property and instance. OSTATE generates models conform to that metamodel from textual or graphical (TOPCASED) AADL descriptions. The OIL metamodel and C language metamodel had to be defined.

In the experimentation, the code was generated from an AADL operational system description. At least, an operational architecture comprises a processor, a memory, a process and a thread, organized in a system (Figure 2). An operational system requires knowledge of the complete minimum execution platform to run the software. Then, it can be instantiated. The instance model is a «flattened» version of the original model. It is also conform to the AADL metamodel (thanks to the instance package) and easier to use because component types and implementations (using inheritance, etc.) are contained in a single instance component.

Figure 2. AADL operational system example

In the experimentation, we focused on the AADL thread component (more specifically, we were interested in their instances), because thread is the base element to consider in real-time systems, and its semantic is better defined than other components in the AADL standard.

2.4 Method

OSEK/VDX tasks are the schedulable units corresponding to AADL threads. Generating code, comply with a standard, from another standard description requires to consider the semantics of the elements from both standards. In the case of schedulable units, we focus on their execution semantics.

OSEK/VDX differentiates extended tasks (which may wait for an event) and basic tasks (which cannot wait for an event) (cf. Figure 3). With only one mode, without fault handling and without considering subprogram calls, the suspended, ready and running states of periodic, sporadic or aperiodic AADL threads and OSEK/VDX tasks follow the same semantics.

Nevertheless, the AADL «awaiting resource» state does not match a particular OSEK/VDX state because of the intended synchronization protocol for shared resources (Immediate Priority Ceiling Protocol). Moreover, the OSEK/VDX «waiting» state may correspond to the AADL «suspend» state for sporadic or aperiodic threads waiting for an activation event.

The initialize and finalize phases, suggested by the AADL standard, are not specified in OSEK/VDX. However, if designers intend to use one of them, its implementation will need to correspond to its expected execution semantics.

Thus, generated OSEK/VDX tasks may respect the AADL thread execution semantics for periodic, sporadic or aperiodic threads. But the transformation designer should be assisted by verification tools to check correspondences between the generated system and the described architecture. To be able to assist designers with tools, or to prove correspondences, the semantics of both standards has to be precise, in the best case formal.

2.5 Results

An AADL system instance with a unique processor corresponds to a CPU in an OIL specification. Various parts of the C code (main function of the application, OSEK/VDX hooks, variable declarations and initializations) can be created either by OIL tools [17] from generated OIL code, or directly by model transformation on AADL system instance. In the experimentation, both approaches were tested.
In the OSEK/VDX standard, all threads executed on the same processor share a virtual address space. Thus, if many threads may be considered, only one process is used. The generated C code (compliant with OSEK/VDX) corresponds to the implementation of dynamic thread semantics (as defined by the AADL standard). It corresponds to a «main program» using specific subprograms. Each subprogram corresponds to a stage in the AADL thread life cycle. The user must develop specific application code to «fill» these subprograms.

In the experimentation, code generation had been performed with ATL model transformations (for the «model to model» phase) and TCS (Textual Concrete Syntax) [12] to generate pretty-printers (for the «model to text» phase). The experimentation lasted two months. The first month was dedicated to becoming familiar with MDa concepts and AMMA tools. The study of the AADL metamodel, the development of the target metamodels (OIL and C language) were achieved in the first part of the second month. The study and the development of ATL transformations (relating to AADL threads) were achieved in the second part of the second month.

The ATL transformations (relating to AADL threads) comprise about three hundred lines for the AADL-to-OIL transformation and about one thousand lines for the AADL-to-C transformation.

The generated OIL and C codes were compiled and used with Trampoline [8], an open source RTOS which, once certified, could be compliant with the OSEK/VDX specification.

3 Experimentation feedback

3.1 The need for alternatives

During the experimentation, implementation choices had to be done due to the gap between the AADL and the OSEK/VDX standards. These choices result in various code generation alternatives.

For instance, if we focus on a particular system composed of periodic threads, a first choice has to be done: they can be implemented with basic or with extended tasks. As shown in figures 4 and 5, an alarm periodically activates both kinds of tasks. But the former is directly activated by the alarm while the latter is activated following the reception of an event.

A second example of choice is the implementation of «initialize», «compute» and «finalize» phases (specified by thread entry point properties): either one task is dedicated to one phase, or a same task executes all phases. These two alternatives imply to analyze the effects of the implementation, considering the intended phase semantics. For example, the former alternative raises the question of «initialize» and «finalize» task priorities.

These two choices are just examples among multiple alternatives. Sequences of choices compose a decision tree. We note that the designer should be assist by decision tools which require to qualify each alternative (memory used, number of events or alarms used, etc.). Choices may be expressed interactively during the transformations. They can also be captured, to define transformation parameters, either
in the AADL model itself or in the form of another model conform to a dedicated «choice description language». The former raises the question of the incorporation of the ability to express alternative in the standard. The AADL «property set» construct is a permissive mechanism that allows to capture choices. But to share knowledge between decision tools, these «property sets» might be integrated in an AADL standard annex. Knowledge capitalization and interchange of both approaches have to be considered.

3.2 Adequacy of MDA for code generation

MDA tools allowed us to benefit from their ability to manipulate metamodels (most of them can be used in the Eclipse development platform). Moreover, MDA tools and declarative languages, such as ATL, simplify the expression of mappings between models. We observe a time saving when we compare with other experimentations that used model pretty-printers to generate code, due to the time to develop the pretty-printer. Thereby, the time spent on the study was dedicated more to thinking about transformations rules than to implementing ATL transformations.

However, ATL transformations are not interactive. This language and associated tools are not well-suited to alternative decisions. Taking into account all possible decisions in one transformation increases transformation size. In the context of MDA, several approaches may be considered for alternatives: using «meta-information» in the source model or in an other model as presented below, chains of transformations, or even using transformations that generate other transformations. The former was experimented in this work, the others will have to be experimented.

4 Conclusion and further work

As from most ADLs which define a precise semantics, code generation from AADL requires a particular attention on the semantics. In this work, generated code must respect the semantics of both AADL and OSEK/VDX. Implying that the former must be compatible with the latter. For this purpose, semantics of both standards have to be precise, in the best case formal, to be able to assist designers with tools, or to prove correspondences.

An ADL provides high-level abstractions to specify systems. Their implementations in a lower level and/or with variant technologies confront designers with alternatives. Some choices have to be done. To do that, designers should be assisted by decision tools which require to qualify each alternative. Choices can be captured either in the ADL system specification itself, with the ADL terms (with AADL properties in the experimentation), or outside the specification in the form of a specific model conform to a dedicated «choice description language». The former raises the question of the incorporation of the ability to express alternatives in the language. Knowledge capitalization and interchange of both approaches, such as the structure of the choices have to be considered. Moreover, MDA allows to consider other approaches, such as transformation chains or using transformations that generate other transformations, to carry out generation process that take account the alternatives. All of these approaches have to be studied.

MDA enables several abstraction levels during the code generation process. Using «pivot» models allows incremental refinement. Thus, several implementation alternatives may be considered. When we compare the presented experimentation with experimentations that divide the generation process into a transformation chain using «pivot» metamodels, it seems that the latter enables to share and to reuse parts of the generation process, contrarily to the former. But other experimentations have to be done to specify these results.

In the presented experimentation, the C code generated from the same model may conform to several coding conventions. To go further, instead of using a specific C language metamodel, we could have used a metamodel defining the common concepts of procedural languages, allowing to retarget code generation from a single model to different languages (such as C or Ada).

At last, the MDA tools experimented are mature enough to address code generation using model transformation. However, the need to manage alternatives during our experimentation revealed that these tools may still be improved. Moreover, transformation tools have to be completed with additional tools such as decision and verification tools.

Acknowledgement

The authors would like to thank Sébastien Faucou and Guillaume Savaton for their advice during the experimentation.

References

[8] J. Bechennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline an open source implementation of the OSEK/VDX...


