A Model-based Framework for Developing Real-Time Safety Ada Systems***

Emilio Salazar, Alejandro Alonso, Miguel A. de Miguel, and Juan A. de la Puente

Universidad Politécnica de Madrid (UPM),
{esalazar,aalonso,mmiguel,jpuente}@dit.upm.es

Abstract. This paper describes an MDE framework for real-time systems with safety requirements. The framework is based on industry standards, such as UML 2.2, MARTE, and the Ada Ravenscar profile. It integrates pre-existing technology with newly developed tools. Special care has been taken to ensure consistency between models and final code. Temporal analysis is integrated in the framework in order to ensure that the real-time behaviour of the models and the final code is consistent and according to the specification.
Automatic code generation from high-level models is performed based on the Ravenscar computational model. The tools generate Ravenscar-compliant Ada code using a reduced set of code stereotypes.
A case study is described for a subsystem of the on-board software of UPMSat2, a university micro-satellite project.

Keywords: Real-time systems, high-integrity systems, model-driven engineering, Ada, Ravenscar profile

1 Introduction

Model-driven engineering (MDE) is a software development approach that allows engineers to raise the abstraction level of the languages and tools used in the development process [17]. It also helps designers isolate the information and processing logic from implementation and platform aspects. A basic objective of MDE is to put the model concept on the critical path of software development. This notion changes the previous situation, turning the role of models from contemplative to productive.

Models provide support for different types of problems: i) description of concepts, ii) validation of these concepts based on checking and analysis techniques, iii) transformation of models and generation of code, configurations, and documentation. Separation of concerns avoids confusion raised by the combination of

* This work has been partially funded by the Spanish Government, project HI-PARTES (TIN2011-28567-C03-01).

** The final version of this paper has been published by Springer-Verlag in LNCS 7896, and is available at link.springer.com.

different types of concepts. Model-driven approaches introduce solutions for the specialization of the models for specific concerns, as well as the interconnection of concerns based on models transformations. It improves communication between stakeholders using the models to support the interchange of information. But the separation of concerns often requires specialized modelling languages for the description of specific concerns.

This paper describes an MDE framework for the development of real-time high-integrity systems. The functional part of the system is modelled using the Unified Modeling Language (UML2) [12]). Real-time and platform properties are added to functional models by means of annotations, using the UML profile for Modelling and Analysis of Real-Time and Embedded Systems (MARTE) [13]). An analysis model for verifying the temporal behaviour of the system using MAST[1] [6] is automatically generated from the MARTE model. Finally, Ada code skeletons are generated, based on the system model and the results of response time analysis. Code generation is based on the Ravenscar computational model [3], and generates Ravenscar-compliant code [18, D.13.1].

Related work includes the Ada code generator in IBM Rhapsody[2] [5], which generates complex Ada code but does not support MARTE or the Ravenscar profile. Papyrus[3] [9], on the other hand, supports functional Ada code generation from UML models, but cannot generate Ravenscar code and does not fully integrate temporal analysis with system models.

The tools developed in ASSERT[4] follow a closer approach. Two sets of tools were developed in this project, one based on HRT-UML [10, 14, 2], and the other one on AADL[5] [7, 8], which later evolved to the current TASTE[6] toolset [15]. Both can generate Ravenscar Ada code and include timing analysis with MAST.

The main differences between these toolsets and the framework presented here are: i) This framework uses up-to date industrial standards such as UML2 and MARTE, instead of ad-hoc adaptions of UML; ii) the transformation tools in this framework have been built with standard languages; iii) the extensive use of standards in this framework makes it possible to use it with different design environments, without being tied to a specific development platform.

The rest of the paper is organised as follows: Section 2 reviews the use of MARTE stereotypes in the framework. Section 3 describes the logical architecture of the framework and the different models that are used in it. Section 4 describes the techniques that are used to generate Ravenscar Ada code. It also includes as a case study some examples from UPMSat2, an experimental microsatellite project which is being carried out at Universidad Politécnica de Madrid (UPM). Finally, some conclusions of the work are drawn in section 5.

---

[1] Modelling an Analysis Suite for Real-Time Applications, `mast.unican.es`

[2] `www.ibm.com/developerworks/rational/products/rhapsody`

[3] `www.papyrusuml.org`

[4] Automated proof-based System and Software Engineering for Real-Time systems, `www.assert-project.net/`

[5] Architecture Analysis Description Language, `http://www.aadl.info`.

[6] The ASSERT Set of Tools for Engineering, `www.assert-project.net/-TASTE-`

2 Modelling real-time systems with MARTE

MARTE is a UML2 profile aimed at providing support for modelling and analysis of real-time and embedded systems [13]. It includes several packages for describing non-functional properties of embedded systems, as well as some secondary profiles for different kinds of systems. This makes MARTE a rather big standard. However, since the framework is aimed at real-time high-integrity systems, only those parts of the MARTE specification that are relevant for this kind of systems are used. The main requirement is to be able to model systems with a predictable behaviour that can be analysed against their specified temporal properties. Such models must be transformed into implementations running on a predictable platform. The Ravenscar computational model [4] is a suitable basis for this purpose.

The modelling elements to be considered are:

- *Input events* describing the patterns for the activation of computations (sequences of actions) in the system, e.g. periodic or sporadic activation patterns.
- *Actions* that have to be executed in response to input events.
- *Precedence constraints and deadlines* for the actions to be executed as a response to an event. Precedence constraints define end-to-end flows of computation that have to be executed within the interval defined by the activation event and the deadline.
- *Resources* needed to execute the actions of the system. Resources can be grouped into *active resources* (e.g. CPUs and networks), and *passive resources* (e.g. shared data). Access to shared resources has to be scheduled in order to guarantee the required temporal properties of the system.

These elements can be described in MARTE using some of its specialized subprofiles. The GQAM (Generic Quantitative Analysis Modelling) profile, which is part of the MARTE analysis model, defines common modelling abstractions for real-time systems. For example, the GaWorkloadEvent stereotype can be used to model input events and the associated timing constraints, and the GaScenario and GaStep stereotypes can be used to specify the response to an event in terms of flows and actions. The SAM (Scheduling Analysis Modelling) profile defines additional abstractions and constraints to build analysable models, including a refined notion of an end-to-end flow.

The resources available for execution can be described using the GQAM:: GaResourcesPlatform stereotype, together with other stereotypes in the GQAM and SAM profiles. Examples of the latter are SaExecHost, SaComm Host, and SharedResource. Scheduling resources are defined with the Scheduler and SecondaryScheduler stereotypes.

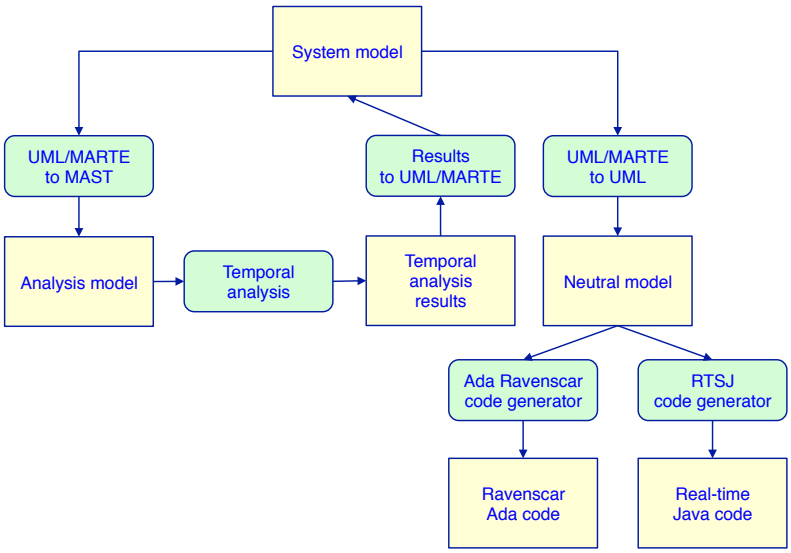

Fig. 1. Architecture of the real-time safety systems development framework

3 A Model-based Framework

3.1 Overview

A model-based framework has been designed in order to provide support for the development of high-integrity real-time systems based on the MDE principles and using UML/MARTE as the main modelling formalism. The overall architecture of the framework is shown in figure 1. Its main elements are four kinds of models:

- **System model**: This is the model that the developer creates using UML and the MARTE profile. It starts as a Platform Independent Model (PIM) that uses MARTE stereotypes to represent the load of the system and the associated real-time attributes (activation patterns, deadlines, etc.). Resources are then incorporated using the appropriate stereotypes to get a platform-specific model (PSM). The model is initially populated with estimates of time attributes, such as blocking times or worst case execution times. Later on, when the actual code is available, these values can be replaced with real measurements, and accurate temporal analysis can be carried out. If the estimates of time attributes are used as requirements for the implementation phase, the results of preliminary analysis based on them should still be valid.
- **Analysis model**: This model is aimed at performing temporal analysis on the system. MAST [6] has been selected as the analysis tool to be used in the framework, as it covers many different situations and analysis methods. The tool can check if the specified time requirements are met, and thus can be used to validate the temporal behaviour of the system early in the

development cycle. Since more accurate execution time measurements are available as the system development advances, the analysis can be repeated as many times as needed.
The analysis model is described using the MAST notation. It is automatically generated from the system model using a transformation tool. The results of the analysis are fed back to the system model by means of another tool, so that the developer can modify the model as needed if the temporal requirements are not met.

- **Neutral model**: This model is intended to simplify code generation for different platforms and programming languages. The model is automatically generated from the system model by transformation tools that have been developed to this purpose, and it is not intended to be read or modified by the user.
  The neutral model is described in plain UML, and has a lower abstraction level than the system model.
- **Implementation model**: The source code for the system is automatically generated from the neutral model. Generator tools for Ada 2005 with the Ravenscar profile and Real-Time Java (RTSJ) have been developed. The Ada generator is further described in section 4. The work on RTSJ is explained in reference [11].

The transformation tools between the above models have been developed by the research team using QVT[7] and MTL.[8]

A more detailed description of the models follows.

3.2 System model

The system model is incrementally built by the developer using UML classes and relations to model the system architecture and its components. MARTE stereotypes are used to define the real-time properties of the relevant classes. Depending on how a class is stereotyped, it can be categorized as a particular real time archetype. The framework recognizes four class archetypes, based on the Ravenscar computational model:

- **Periodic**. Instances of a periodic class execute an action cyclically, with a given period. An offset may be specified for the first execution. Each execution has a fixed deadline with respect to its activation time.
- **Sporadic**. Sporadic objects execute an activity on each occurrence of some activation event. As above, a deadline is defined relative to the activation time.
  Periodic and sporadic classes are *active* classes. Their activation patterns and deadlines are defined using the GQAM::GaWorkloadEvent stereotype with a

[7] Query/View/Transformation, `www.omg.org/spec/QVT/`
[8] Model to Text Transformation Language, `http://www.omg.org/spec/MOFM2T/1.0/`

periodic/sporadic arrival pattern and a deadline. Scheduling details are defined when appropriate in the design process using the GRM::SchedulableResource stereotype. A fixed-priority preemptive scheduling policy is assumed by default.

- **Protected**. Protected objects encapsulate shared data that is accessed in mutual exclusion. A protected class is defined with the GRM::MutualExclusionResource stereotype.
- **Passive**. Passive objects have no real-time properties and are not used by more than one active object. Classes without any MARTE stereotypes are characterized as passive.

3.3 Analysis model

The GRM, GQAM and SAM MARTE profiles are designed for the automatic generation of schedulability analysis models. These models can be generated and analysed at early modelling phases, so that design decisions can be made depending on the temporal behaviour of the system.

The MARTE analysis annotations are represented with UML extensions. In practice, the analysis model only depends on the UML specification of sequence behaviours. These extensions include references between them, and all together define an analysis model. A UML model may include as many analysis scenarios as SAM::SaAnalysisContext stereotype applications.

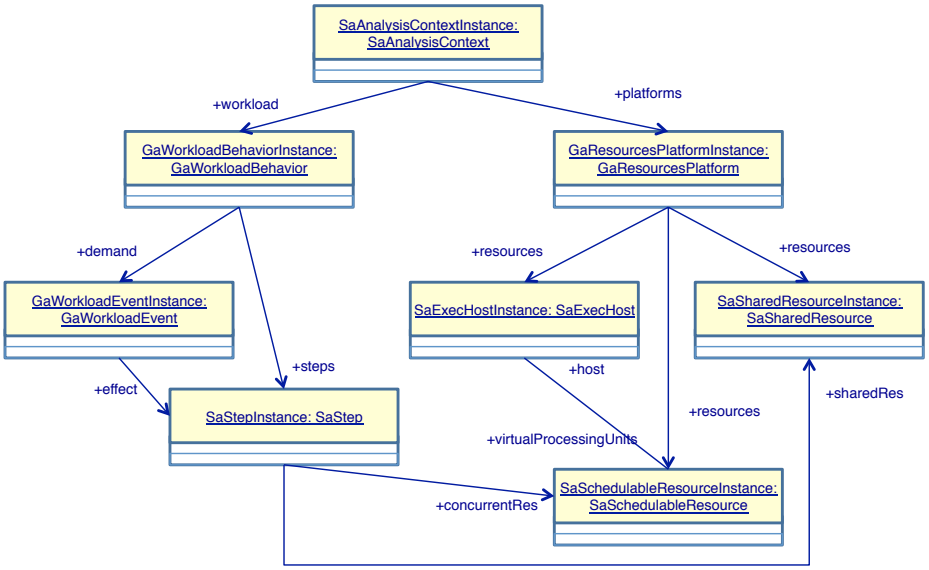

Fig. 2. General structure of an analysis model

Figure 2 shows some relations between analysis stereotypes that summarize the general structure of the models. The root is the analysis context (typically a package or a model; alternative solutions can include several analysis contexts). It identifies the set of workload behaviours and platform resources.

A workload behaviour is associated to a workload event and to the sequence of steps and scenarios that are executed when the event occurs. Notice that sub-scenarios can also be specified as an effect of a workload event. Steps are associated to schedulable resources.

Platform resources include schedulable resources, executable resources, and mutual exclusion resources. Schedulable resources define a flow of execution of steps. They are associated with a processor and a scheduler.

In summary, the analysis model is based on four basic concepts:

- *Specification of load events in the system.* These are the sources of load in the system. The arrival pattern of events must be specified in order to analyse the temporal behaviour of the system.
- *Event responses.* A sequence of steps defining behaviour associated to event occurrences. The required information includes the precedence relations between events and the resources needed to execute them, including timing data (e.g. execution time budgets).
- *Resources.* Steps are executed in the context of schedulable resources (e.g. threads, processes or tasks). Scheduling parameters, such as priorities, must be also defined.

  Some additional resources may also be required. Examples of different kinds of resources include computing resources (e.g. processors), communication resources, synchronisation resources, and mutual exclusion resources. All of them require some parameters to be defined for temporal analysis (e.g. ceiling priorities for mutual exclusion resources).
- *Schedulers.* Schedulers define the rules for sharing resources among schedulable resources. For example, fixed-priority is a well-known scheduling method for computing resources.

All of these analysis elements can be modelled in MARTE, using the stereotypes mentioned in section 2 above. The MARTE model is translated into the input language of the MAST analysis toolset[9] by means of a transformation tool. The tools make use of different schedulability analysis methods to compute temporal data such as worst case response time for events and steps, occupation of resources, and optimal protocol and scheduling parameters for resources.

The code generated in Ada generator must be consistent with the results of the scheduling analysis results. To this purpose, the results are fed back to the system model in order to fix any inconsistencies and provide a feasible description of the system to the neutral model.

[9] See `mast.unican.es` for details on the analysis tools.

3.4 Neutral model

In order to implement code generation in a flexible and efficient way, a neutral model is used as an intermediate step between the system model and the final code. Since the framework is focused on high-integrity systems, the code has to be restricted according to appropriate profiles in order to ensure that it runs in a predictable way and its timing behaviour complies with the specification. The use of profiles simplifies code generation, which can be based on a common notation independent of the programming language to be used.

The neutral model is defined in plain UML, without using any MARTE stereotypes. Only information that is relevant for code generation is included in this model. Language-dependent elements are avoided, in order to enable code generation for different implementation languages.

The driving principles in the generation of the neutral model are:

- *Include only data that is needed for code generation*, e.g. period, phase, priority. The system model may include other kinds of information, which are not needed for this purpose. This rule simplifies the implementation of the code generator, and increases its efficiency.
- *Keep data types as simple as possible*, in order to reduce the semantic gap between UML and the implementation languages. The neutral model uses mostly simple data types (e.g. natural, integer, string), and tries to avoid the use of complex data types. In particular, custom MARTE data types, which would be difficult to translate into a specific programming language, are excluded.
- *Keep the model independent of the target programming language.* Indeed, the main goal of using an intermediate model is to be able to generate code for different programming languages.
- *Support traceability between the system model and the final code and vice versa*, in order to make it possible to indicate which part of code corresponds to which part of the system model at any time in the development process. This also includes the temporal analysis results, which should also be traceable in order to identify the source of scheduling-related constructs in the code. In this way, if a problem arises in the final code, the original model element that causes the error can be quickly identified and corrected as needed.

The neutral model is built from a small number of common real-time patterns matching the archetypes described in section 3.2 These patterns are represented by UML plain classes with additional annotations including all the required data coming from the original system model that cannot be expressed in UML.

The most relevant kinds of annotations include:

- WCET, for the worst-case execution time of an activity.
- Deadline error handler. Defines the user code to be executed when a deadline overrun occurs.
- General exception handler. Defines the code to be executed when an exception is raised.

– Last chance exception handler. Defines the user code to be executed as "last wishes routine", before terminating a program.
– Task initialization. Defines some code to be executed at system start time by a periodic or sporadic object.

4 Ravenscar Ada Code generation

4.1 Ada generation overview

This section describes the generation of Ada source code from the neutral model, which only includes the necessary information for this purpose, in a language-independent way. The neutral model can be used for generating code in Ada, RTSJ, or any other language suitable for real-time systems.

The Ada code generator relies on international industrial standards. It has been developed using QVT and MTL, as mentioned in 3.1 above. The neutral model is described with plain standard UML 2.2, and the output is Ada 2005 with the Ravenscar Profile restrictions [18]. The generator produces Ada code skeletons with a temporal behaviour consistent with the system model, including the results of temporal analysis as previously described in section 3.3. This approach facilitates the link with functional code.

Some aspects of the code generation process are illustrated with fragments of the Attitude Determination and Control Subsystem (ADCS) subsystem of the UPMSat2 on-board software system[1].

4.2 Code generation for components

The top level description of the system is based on UML components, which are composed of a set of classes. UML components include an interface, and a set of classes that implement the public operations and the component functionality. The interface defines the component contract with the client, which specifies its public functionality. It is mapped into an Ada package. Its specification includes the signature of the operations that are exported from the interface of the UML component. The corresponding body simply redirects the exported operation to the corresponding internal package operation. This approach follows software engineering principles, such as information hiding, loose coupling, and facilitates code generation and maintenance for different execution platforms. Figure 3 shows the external view of the ADCS component, as well as its internal structure.

The internal classes of the component are mapped into private packages, as described below, according to their real-time archetypes: periodic, sporadic, protected or passive activities. Hierarchical packages are used for representing the structure modelled with the UML components.

The Ada code generated for the interface is shown in figure 4. This component exports three methods that are mapped into Ada operations in the package specification. In the body, these operations call the corresponding internal implementation method.

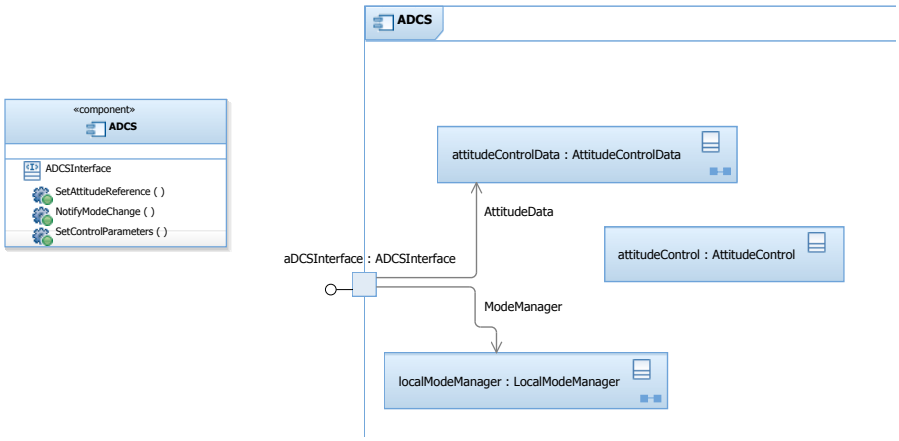

Fig. 3. Attitude Determination and Control Subsystem UML component

```
with ADCS.BasicTypes; use ADCS.BasicTypes;
package ADCS.Interfaces is
   procedure NotifyModeChange     (mode : in Mode_Type);
   procedure SetAttitudeReference (ref  : in Reference_Type);
   procedure SetControlParameters (conf : in Configuration_Type);
end ADCS.Interfaces;
```

```
with ADCS.LocalModeManager;
with ADCS.AttitudeControlData;
package body ADCS.Interfaces is

  procedure NotifyModeChange (mode : in Mode_Type) is
  begin
    ADCS.LocalModeManager.LocalModeManager.SetMode (mode);
  end NotifyModeChange;

  procedure SetAttitudeReference (ref : in Reference_Type) is
  begin
    ADCS.AttitudeControlData.AttitudeControl.SetActitudeReference(ref);
  end SetAttitudeReference;

  procedure SetControlParameters (conf : in Configuration_Type) is
  begin
    ADCS.AttitudeControlData.AttitudeControl.SetControlParameters(conf);
  end SetControlParameters;

end ADCS.Interfaces;
```

Fig. 4. Generated package for an UML component interface

Fig. 5. Design view of the ADCS classes

Figure 5 shows a detailed view of the internal classes belonging to the ADCS subsystem. It can be noticed that the MARTE stereotypes are printed in the upper part of the corresponding classes.

The *ADCS.AttitudeControl* class implements the control algorithm for keeping a given attitude for the satellite. It is a periodic entity, and its graphical representation shows the annotations GRM::SchedulableResource and GQAM::WorkloadEvent.

There are two protected entities, which are stereotyped with the GRM::MutualExclusionResource annotation: *ADCS.AttitudeControlData* and *ADCS.LocalModeManager*. One is in charge of keeping information for the control algorithm. The other one is used for managing the operational mode of the component.

4.3 Code generation for classes

Code generation is based on a set of code templates, which are directly related with the archetypes in the neutral model. As mentioned above, classes in this model are annotated with its related archetype. Currently, four archetypes are supported: Periodic, Sporadic, Protected, and Passive.

There are several suitable implementations of these archetypes in the literature. This work is based on the Ravenscar Ada generic packages defined in [16]. Some additional features have been added, for including issues such as deadline overrun handling, WCET overrun handling, Ada standard exceptions handling,

user defined task initialization, or user-defined stack size. These features were included in order to enable dealing with more general tasking models. In the context of this work, some of them are disabled by default, as they are not Ravenscar-compliant.

Each class in the neutral model is represented by an Ada package. The package includes in its private part an instance of a generic package where the class archetype is defined. All the required dependencies are included in the package description.

Class dependencies in the neutral model are converted into Ada with clauses. There are three different types of dependencies:

- Explicit user-defined dependencies. These dependencies are explicitly defined in the system model, and they are directly translated, without any additional processing.
- Implicit dependencies due to attributes or parameter types. If an attribute or a parameter is defined as a non-primitive type (e.g. attributes which are instances of other classes), the generator automatically adds the required dependencies.
- Implicit dependencies due to automatically generated code. Several Ada features (e.g. last chance exception handler, etc.) have dependencies on other Ada packages. They are also automatically added when they are required.

With respect to the internals of the package, the main action is the instantiation of the generic package that corresponds to the class archetype, which is carried out in the private part of the package. The parameters needed for the generic instantiation depend on the type of the archetype. In the case of active classes, the following parameters are required:

- Real-time parameters: priority, period or minimal inter-arrival time, and initial offset.
- Functional code parameters: periodic or sporadic activity, and initialization procedure.
- Error handling parameters: procedures for dealing with timing related exceptions and with Ada standard exceptions.

The code listed in figure 6 shows an example of a specification file generated for a periodic activity. The instantiation call is located at the end of the private part.

Protected classes, annotated as mutual exclusion resources, are translated into a package that includes a protected object. The stereotype allows to define the ceiling priority for the object, which is directly translated into the corresponding parameter in Ada. The code generated for the *ADCS.AttitudeControlData* class is shown in figure 7

Finally, the framework supports two types of passive classes, in order to provide additional flexibility to the developer:

- *Singleton class:* If the developer annotates an UML class as singleton, there may only be one instance of it in the system. In this case, the code generator

```
with Ada.Real_Time.Timing_Events;
use  Ada.Real_Time.Timing_Events;
with GNAT.IO;                          use GNAT.IO;
with Ada.Exceptions;                   use Ada.Exceptions;
with uml2ada.exceptions;               use uml2ada.exceptions;
with uml2ada.periodic_tasks;

package example.examplePlatform.subsystem1.OBDH.ADCS.PeriodicADCS is

   PeriodicADCS_priority : constant := 8;
   PeriodicADCS_period   : constant := 9854;
   PeriodicADCS_offset   : constant := 234000000;
private

   procedure Activity;

   procedure Activity_Initialization;

   protected DeadlineHandler is
      procedure DeadlineErrorHandler (Event : in out Timing_Event);
   end DeadlineHandler;

   procedure ConstraintErrorHandler (e : in Exception_Occurrence);

   package PeriodicADCS_periodic_task is new
     uml2ada.periodic_tasks (
      Priority                 => PeriodicADCS_priority,
      Period                   => PeriodicADCS_period,
      Offset                   => PeriodicADCS_offset,
      Periodic_Activity        => Activity,
      Initialization           => Activity_Initialization,
      Deadline_Ovr_Handler     => DeadlineHandler.DeadlineErrorHandler'Access,
      Constraint_Error_Handler => ConstraintErrorHandler'Access,
      Program_Error_Handler    => Default_Exception_Handler,
      Storage_Error_Handler    => Default_Exception_Handler,
      Tasking_Error_Handler    => Default_Exception_Handler,
      Other_Error_Handler      => Default_Exception_Handler);

end example.examplePlatform.subsystem1.OBDH.ADCS.PeriodicADCS;
```

Fig. 6. Specification of a generic periodic archetype

```
with ADCS.BasicTypes; use ADCS.BasicTypes;
private package ADCS.AttitudeControlData is

   protected AttitudeControl is
      pragma Priority (10);

      procedure SetActitudeReferece (reference : in Reference_Type);
      procedure SetControlParameters (config : in Configuration_Type);
      function GetControlParameters return Configuration_Type;

   private
      internal_configuration : Configuration_Type;
      internal_reference     : Reference_Type;
   end AttitudeControl;
end ADCS.AttitudeControlData;
```

Fig. 7. Specification of a generated protected object

produces a package with a public interface that is composed by a set of operations.

- *Standard passive class:* Multiple instances of this classes are allowed. The Ada code generator produces a package that implements an abstract data type. The public interface includes the type definition and its primitive operations.

4.4 Code generation for methods

The methods specified in classes of the neutral model are translated into Ada subprograms according to the following rules:

a) Each UML method is translated into an Ada procedure or function:
   - Methods with a return parameter are translated as functions.
   - Methods without a return parameter are implemented as procedures.
   - Parameters specified as in, out, and inout are translated into their Ada equivalents.
   - Methods that are declared as private in the *neutral* model are generated in the private part of the Ada specification. Otherwise, they are placed in the public part.

b) UML attributes and method parameter types are implemented as Ada types.
   - Integer, Positive and Boolean primitive types are directly implemented by the corresponding Ada types.
   - The implementation of the String primitive type is a little more difficult since Ada strings must be constrained at compilation time. Unconstrained string parameter types are thus translated into the library Unbounded_String type instead of the Ada String type. Consequently, a dependence on the package Ada.Strings.Unbounded has to be added.
   - Enumerations are translated into Ada enumeration types.

   Parameters can also be defined as types of model-defined classes or as constrained arrays.

c) Initialization of primitive or enumeration type attributes is also supported. An initialized string attribute results in an Ada String, since its length is known at compilation time.

d) Standard Ada exceptions raised in the functional code are, by default, propagated. Nevertheless, is possible to provide user-defined handlers for such exceptions. A handler for user-defined exceptions can also be provided.

e) It is also possible to provide a user-defined last-chance exception handler in order to execute a "last wishes" routine if the system unexpectedly terminates.

f) By default, periodic activities raise a Program_Error exception in case of a deadline overrun. However, a user-defined routine can be specified to be executed instead.

5 Conclusions

Model-driven Engineering allows developers to raise the abstraction level of software design, so making the development process safer and faster. At the implementation side, Ada and the Ravenscar profile provide excellent support for building predictable real-time systems that can be statically analysed for a specified temporal behaviour. The work described in this paper has been directed at combining the best of both worlds through the use of a specialised framework covering the design and implementation development phases of high-integrity real-time systems.

The main contributions of the framework are its alignment with industrial standards, specifically OMG standards and Ada, and the tight integration of the system model with the analysis model. Moreover, the strict adherence to UM2/MARTE standards and the use of standard tools make it possible to implement the framework on a variety of tools, without depending on a particular toolset. Most of the transformation tools described in the paper have been implemented and tested on IBM RSA (Rational Software Architect), but migrating to other environments (e.g. Eclipse) can be done with comparatively little effort. The transformation tools are freely available at `www.dit.upm.es/str`.

The use of neutral model facilitates generating code for different programming languages. Generators for Ada and Real-Time Java have been implemented and can be found at the same location as the transformation tools.

Future work includes enhancing the transformations between the system model and the analysis model, which are now at an early stage of development, and adding support to include functional code generated from other tools (e.g. Simulink) into the real-time skeletons generated by the framework.

Acknowledgments. The framework described in this paper was originally developed within the European project CHESS,and has been completed and extended in the HIPARTESproject. We would like to acknowledge the financial support of the European Commission FP7 program and the Spanish national R+D+i plan as well as the collaboration with the partners in both projects.

References

1. Alonso, A., Salazar, E., de la Puente, J.A.: Design of on-board software for an experimental satellite. In: Jornadas de Tiempo Real — JTR-2013 (2103), `www.dit.upm.es/~str/papers/pdf/alonso&13a.pdf`, available at `www.dit.upm.es/~str/papers/pdf/alonso&13a.pdf`
2. Bordin, M., Vardanega, T.: Correctness by construction for high-integrity real-time systems: A metamodel-driven approach. In: Abdennadher, N., Kordon, F. (eds.) 12th International Conference on Reliable Software Technologies — Ada-Europe 2007. pp. 114–127. No. 4498 in LNCS, Springer-Verlag (2007)
3. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar tasking profile for high integrity real-time programs. In: Asplund, L. (ed.) Reliable Software Technologies — Ada-Europe'98, Lecture Notes in Computer Science, vol. 1411, pp. 263–275. Springer Berlin Heidelberg (1998)

4. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar profile in high integrity systems. Ada Letters XXIV, 1–74 (June 2004)
5. Gery, E., Harel, D., Palachi, E.: Rhapsody: A complete life-cycle model-based development system. In: Butler, M., Petre, L., Sere, K. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 2335, pp. 1–10. Springer Berlin Heidelberg (2002)
6. González Harbour, M., Gutiérrez, J.J., Palencia, J.C., Drake, J.M.: MAST modeling and analysis suite for real time applications. In: Proceedings of 13th Euromicro Conference on Real-Time Systems. pp. 125–134. IEEE Computer Society Press, Delft, The Netherlands (June 2001)
7. Hamid, I., Najm, E.: Operational semantics of Ada Ravenscar. In: Kordon, F., Vardanega, T. (eds.) Reliable Software Technologies – Ada-Europe 2008, pp. 44–58. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2008)
8. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Tr. Embedded Computer Systems 7(4), 1–25 (2008)
9. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus UML: an open source toolset for MDA. In: Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009). pp. 1–4 (2009)
10. Mazzini, S., Puri, S., Vardanega, T.: An MDE methodology for the development of high-integrity real-time systems. In: Design, Automation and Test in Europe, DATE 2009. pp. 1154–1159. IEEE (2009)
11. de Miguel, M.A., Salazar, E.: Model-based development for RTSJ platforms. In: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems. pp. 175–184. JTRES '12, ACM, New York, NY, USA (2012)
12. OMG Unified Modeling Language (UML) (2011), `http://www.omg.org/spec/UML/2.4.1/`, version 2.4.1
13. OMG UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (2011), `http://www.omg.org/spec/MARTE/`, version 1.1
14. Panunzio, M., Vardanega, T.: A metamodel-driven process featuring advanced model-based timing analysis. In: Abdennadher, N., Kordon, F. (eds.) Reliable Software Technologies – Ada Europe 2007, Lecture Notes in Computer Science, vol. 4498, pp. 128–141. Springer Berlin Heidelberg (2007)
15. Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T., Hugues, J.: The TASTE toolset: Turning human designed heterogeneous systems into computer built homogeneous software. In: 5th Int. Congress on Embedded Real-Time Software and Systems — ERTS2 2010 (May 2010)
16. Pulido, J., de la Puente, J.A., Bordin, M., Vardanega, T., Hugues, J.: Ada 2005 code patterns for metamodel-based code generation. Ada Letters XXVII(2), 53–58 (August 2007), proceedings of the 13th International Ada Real-Time Workshop (IRTAW13)
17. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2) (2006)
18. Taft, S.T., Duff, R.A., Brukardt, R.L., Plöedereder, E., Leroy, P. (eds.): Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652:1995/Amd 1:2007. No. 4348 in Lecture Notes in Computer Science, Springer-Verlag (2006)