Automatic Ada Code Generation Using a Model-Driven Engineering Approach*

Diego Alonso, Cristina Vicente-Chicote, Pedro Sánchez, Bárbara Álvarez, and Fernando Losilla

División de Sistemas e Ingeniería Electrónica (DSIE)
Universidad Politécnica de Cartagena, Campus Muralla del Mar, E-30202, Spain
{diego.alonso,cristina.vicente,pedro.sanchez,balvarez}@upct.es

Abstract. Currently, Model-Driven Engineering (MDE) is considered one of the most promising approaches for software development. In this paper, a simple but complete example based on state-machines will be used to demonstrate the benefits of this approach. After defining a modelling language (meta-model) for state-machines, a graphical tool will be presented which is aimed at easing the description and validation of state-machine models. These models will then be used as inputs for another tool which will automatically generate the corresponding Ada code, including a simulation program to test the correctness and performance of the implemented application.

1 Introduction

In the last decades, programming languages and *Computer Aided Software Engineering* (CASE) tools promised an important improvement in the way software was developed. This improvement was due to the increase of the level of abstraction provided by the languages and tools used for software development. However, there have been several factors that have led to lower benefits than expected, such as: (1) the lack of accuracy of the used notations and, as a result, the loss of very relevant attributes (e.g. safety, reliability, etc); (2) the strong dependency of software on the execution infrastructure, i.e. the use of code-oriented designs; and (3) the impossibility of reusing most of the developed software artefacts in other projects (except when applying design patterns [9]). The main reason why all of these tools have failed to accomplish their promises can be summarized in the following sentence: all of them provide higher levels of abstraction in the *solution space* rather than in the *problem space*.

Model-Driven Engineering (MDE) is an emerging paradigm aimed at raising the level of abstraction during the software development process further than third-generation programming languages can. MDE technologies offer a promising approach to address the inability of third-generation languages to cope with increasing software complexity, allowing designers to describe domain concepts effectively [13]. This new paradigm uses models as first-class artefact's, making it possible to model those concepts needed to fully describe new systems, together with the relationships existing

* This work has been partially funded by the Spanish CICYT project MEDWSA (TIN2006-15175-C05-02) and the PMPDI-UPCT-2006 program (Universidad Politécnica de Cartagena).

N. Abdennahder, F. Kordon (Eds.): Ada-Europe 2007, LNCS 4498, pp. 168–179, 2007.

between them. Objects are replaced by models, and model transformations appear as a powerful mechanism for incremental and automatic software development [7].

The benefits of raising the level of abstraction from code blocks (either functions or classes) to models are clear. When generative techniques become more mature, MDE will exhibit all its potential for automating code generation, while keeping the final system compliant with the original requirements, following the correct-by-construction philosophy [5]. For this reason, MDE can be considered a very promising approach, especially in those domains where certain requirements must be guaranteed, a traditional field for Ada applications.

1.1 The Model-Driven Approach

MDE promotes a software development process centred on models which are systematically used everywhere. In this approach, models play a central role guiding not only the software development and documentation processes but also its management and evolution. In MDE [12] models are created starting from formal meta-models, which may describe complementary views of the same system, observed at different abstraction levels. The use of formal meta-models allow designers to build both models and transformations between them in a more natural way.

The *Model Driven Architecture* (MDA) proposal [1] is the particular view of the MDE process proposed by the *Object Management Group* (OMG). MDA defines a software development process aimed at separating the business logic from the technological platform. To achieve this, MDA proposes three modelling abstraction levels. Firstly, a *Computation Independent Model* (CIM) represents the system seen as a business process. Secondly, this CIM is refined into a *Platform Independent Model* (PIM) which describes different aspects of the CIM in more detail but which does not contain information about any specific execution platform. This PIM can evolve, through model transformations, to other more specific PIMs. Finally, when a PIM can not further evolve without specifying certain platform-dependent details, it evolves to one or more *Platform Specific Models* (PSMs), one for each platform being considered for system deployment. Each PSM can then evolve independently to other more specific PSMs until the final application code can be automatically generated.

The OMG has defined a series of standards to support MDA and to achieve interoperability among all the tools involved in the software development process defined by this approach. Among these standards, it is worth highlighting the *Meta-Object Facility* (MOF) [2] and the *XML Metadata Interchange* (XMI) standards. The MOF specification defines a meta-language and a set of standard interfaces aimed at defining and manipulating both models and meta-models. The XMI specification enables to store MOF artefact's into XML files, so they can be freely interchanged between tools that conform to these two standards. Although both MOF and XMI have been defined to support the MDA approach, they can also be used in the more general MDE approach, as it will be shown in this paper.

1.2 Goals of the Paper

The main goal of this paper is to highlight the advantages of the MDE approach by means of a complete example, i.e. from the meta-model definition, to the implementation

of a graphical modelling tool and a model transformation to enable automatic Ada code generation. To achieve this goal, the following sub-goals will be addressed:

- The first step when using a MDE approach is to define the modelling language (meta-model) which should include those concepts relevant to the application domain being considered. We have chosen to define a simplified version of the UML 2.0 state-machine meta-model, since these artefacts are quite simple and well known in the real-time community. This sub-goal will be covered in section 2.
- A graphical modelling tool, based on the previously defined meta-model, has been implemented in order to help designers to build new state-machine models. This tool, together with a couple of example state-machines built with it, will be presented in subsection 3.1.
- Finally, the Ada code corresponding to one of the example state-machines will be automatically obtained using a model-to-text (M2T) transformation. This transformation will be described in subsection 3.2.

After covering these goals, the paper will present some related work together with some conclusions and future research lines.

2 A Motivation Example: Modelling State-Machines

This section presents an example based on the description of state-machines, which will be used through the rest of this paper to illustrate the benefits of the MDE approach. We have chosen state-machines since they are quite simple and widely used to describe high-integrity and safety critical systems, well-known application domains for the Ada community.

Applying a MDE approach to model state-machines (or any other general-purpose or specific application domain) requires selecting or defining the most appropriate modelling language (meta-model) for describing them. As stated in section 1, the OMG currently offers a set of standards related to MDA, which include MOF as the top level meta-meta-modelling language (language for describing meta-models, e.g. UML).

Nowadays, the most widely-used implementation of MOF is provided as an Eclipse[1] plug-in called *Eclipse Modelling Framework* (EMF) [6]. Although EMF currently supports only a subset of MOF, called *Essential MOF* (EMOF), it allows designers to create, manipulate and store (in XML format) both models and meta-models. Actually, many other MDE-related initiatives are currently being developed around EMF, such as *Graphical Modelling Framework* (GMF), *EMF Technology* (EMFT) or *Generative Modelling Technologies* (GMT).

All meta-models designed using EMF look very much like UML class diagrams where: (1) domain concepts are represented using boxes (*EClass* in EMOF), (2) inheritance arrows define concept specialisation (EMOF supports multiple inheritance),

[1] Eclipse is an open source, platform-independent software framework both for developing software (like a classical development environment) and for deploying final applications (what the project calls "rich-client applications"). It is available at `http://www.eclipse.org`

and (3) association arrows represent relationships between concepts (*EReference* in EMOF) with or without containment (composition). All these EMF elements will be shown in the state-machine meta-model which is presented in the following subsection.

In this paper a simplified version of the UML 2.0 state-machine meta-model has been chosen to illustrate the power and benefits of the MDE approach. Thus, some of the concepts currently included in the original UML 2.0 state-machines have been removed for the sake of simplicity, e.g. regions and certain kinds of pseudo-states (join, fork, choice, etc.).

As shown in Fig. 1, the simplified state-machine meta-model contains a set of `vertices` and `transitions`. Two different kinds of `vertices` can be defined: `states` and `pseudo-states`. The difference between them is quiet subtle: although a state-machine can only be in a certain observable `state` at a time and never in a `pseudo-state`, pseudo-states are needed to fully describe state-machine behaviour, e.g. defining the initial pseudo-state as the starting execution point of the state-machine. Conversely, a `finalstate` is observable and thus it should be considered a state, more specifically, the state where the state-machine execution ends.

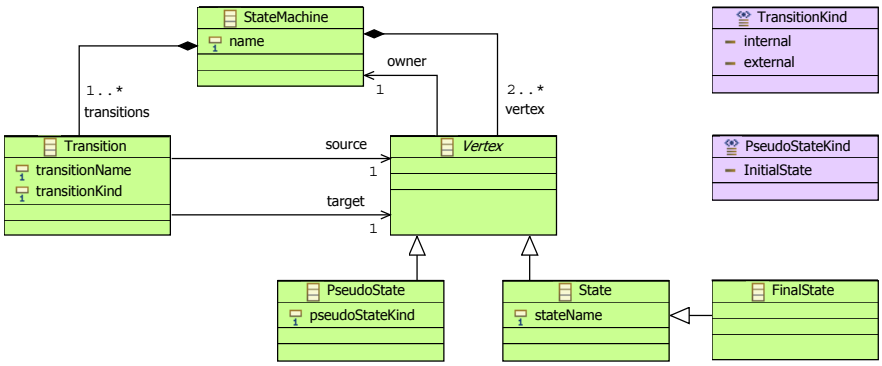

Fig. 1. The state-machine meta-model

Two different kinds of transitions have been included in the meta-model. On the one hand, external transitions exit the current state (calling its *onExit* activity) and, after executing their *fire* activity, they enter the same or another state (calling the corresponding *onEntry* and *do* activities). On the other hand, internal transitions do not change the current state. They only execute their *fire* activity and make the current state call its *do* activity (neither the *onEntry* nor the *onExit* activities are executed).

Transitions are not triggered by *events* and they have no *guards* (actually, these two concepts have not been included in the meta-model). Conversely, transitions are triggered by their names; this is why, in order to obtain a deterministic behaviour, all transitions leaving from the same state must have different names.

3 Tools for Modelling and Implementing State-Machines

This section presents the two tools implemented as part of this work, both of them based on the state-machine meta-model previously described. First, we describe a graphical modelling tool which allows designers to create and validate new state-machine models in a very intuitive way. The section ends showing a tool for automatically generating Ada code from any state-machine graphical model correctly built (and validated) with the previous tool.

3.1 A State-Machine Graphical Modelling Tool

A state-machine graphical modelling tool has been implemented using the facilities provided by the Graphical Modelling Framework (GMF) Eclipse plug-in. This tool is aimed at developing graphical model editors from any EMF meta-model. GMF integrates the facilities provided by other Eclipse plug-ins, namely: Graphical Editing Framework (GEF) and EMFT-OCL. The first of these plug-ins enables the definition of graphical elements that will represent the concepts included in the meta-model, while the second one enables the evaluation of OCL [4] queries and constraints in the models depicted using the GMF tool.

The steps of the process and the elements required to build a graphical modelling tool using GMF are illustrated in Fig. 2. Firstly, the EMF meta-model (*.ecore* file) must be defined using either the basic EMF tree-editor or the graphical meta-modelling tool provided with GMF (see the GMF ecore diagram shown in Fig. 1). Secondly, a set of graphical elements that intuitively represent the concepts included in the meta-model must be created (*.gmfgraph* file). Then, a tool palette has to be designed to allow the user to create each graphical element with the corresponding tool (*.gmftool* file). Finally, a mapping between the elements defined in the three previous files must be described, i.e. the domain concepts included in the meta-model must be mapped to its graphical representation and to the corresponding creation tool in the palette.

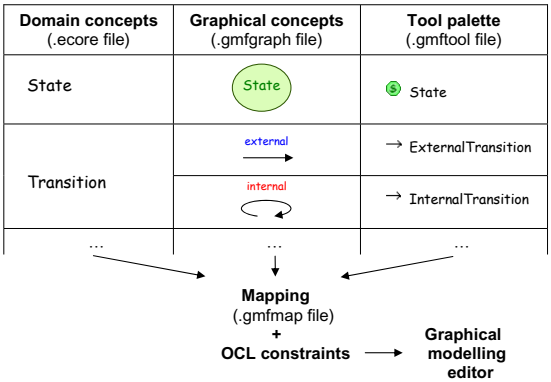

Fig. 2. Sketch of the process and the elements required for building a GMF tool

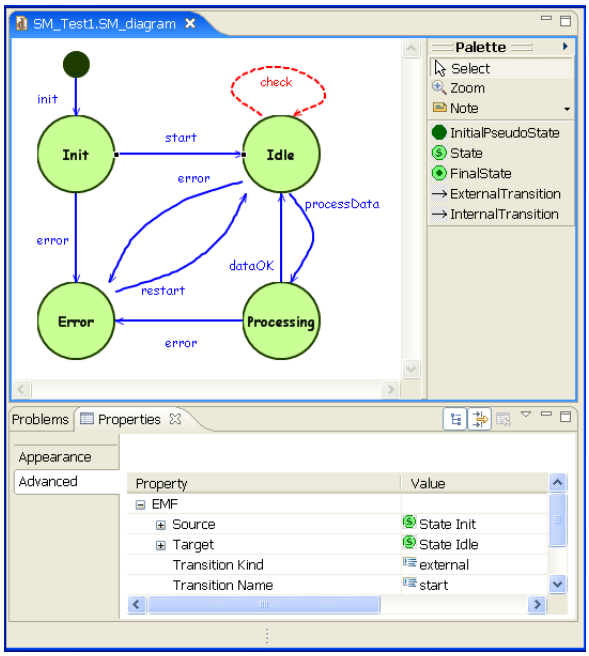

Fig. 3. A graphical state-machine model correctly validated

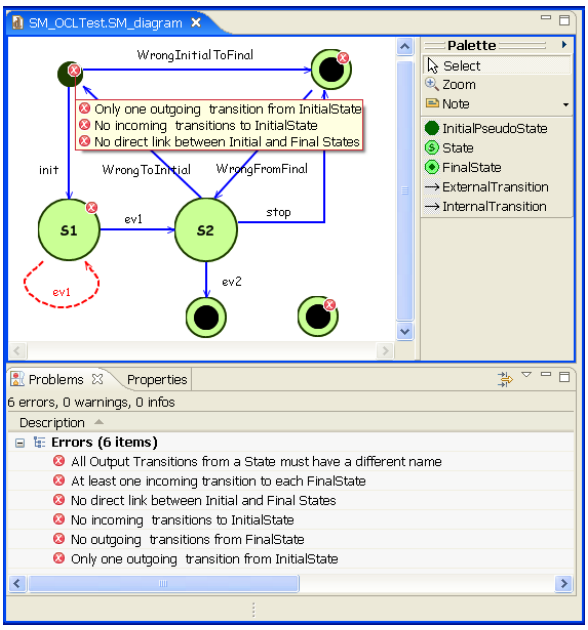

Fig. 4. Another graphical state-machine model. In this case, the validation process detects some incorrect elements (marked with small crosses).

Only the association relationships included in EMF meta-models can be somehow restricted by defining their upper and lower bounds. Actually, there are no means to include any further restrictions or any semantics into a meta-model using only EMF. However, GMF allows designers to define what should be considered a correct model and what should not, according not only to the meta-model but also to some OCL constraints defined in the mapping file.

The constraints included in our meta-model are very similar to those defined for the UML 2.0 state-machines. Some of these constraints are: (1) `Initial pseudo-states` have one and only one outgoing and no incoming `Transitions`; (2) `FinalStates` can not have any outgoing `Transition`; (3) All outgoing `Transitions` from a certain `State` must have different names; etc. As an example, the OCL code for testing the last constraint has been defined as follows:

```
self.owner.transitions -> forAll ( t1, t2 | (( t1.source = self )
    and ( t2.source = self ) and ( t1.target <> t2.target ))
    implies ( t1.transitionName <> t2.transitionName )
)
```

Next, two state-machine models built using the GMF graphical modelling tool implemented as part of this work are presented. Fig. 3 shows a valid state-machine model, while Fig. 4 illustrates another model that has not been correctly validated according to the OCL constraints defined in the GMF mapping file. Actually, as shown in the ”Problems” tab under the diagram depicted in Fig. 4, six OCL constraints are violated, e.g. the following three errors appear associated to the initial state: (1) it can only have one outgoing transition (*init* or *WrongInitToFinal* has to be removed), (2) it can not have any incoming transition (*WrongToInitial* has to be removed), and (3) it can not be directly linked to a final states (*WrongInitToFinal* has to be removed).

The state-machine model shown in Fig. 3 will be used to obtain the corresponding Ada code with the model-to-text transformation described in subsection 3.2. Part of the XML file corresponding to this state-machine model is shown below.

Excerpt of the XML code corresponding to the state-machine model shown in Fig. 3:

```
 <?xml version="1.0"
encoding="UTF-8" ?>
  <StateMachineTool:StateMachine xmi:version="2.0"
     xmlns:xmi="http://www.omg.org/XMI"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:StateMachineTool="StateMachineTool" name="SMExample">
    <transitions transitionName="init"
      source="//@vertex.0" target="//@vertex.1"
      transitionKind="external"/>
      ...
    <transitions transitionName="check"
      source="//@vertex.3" target="//@vertex.3"
      transitionKind="internal"/>
    <vertex xsi:type="StateMachineTool:PseudoState"/>
    <vertex xsi:type="StateMachineTool:State" stateName="Init"/>
    <vertex xsi:type="StateMachineTool:State" stateName="Error"/>
    <vertex xsi:type="StateMachineTool:State" stateName="Idle"/>
    <vertex xsi:type="StateMachineTool:State" stateName="Processing"/>
  </StateMachineTool:StateMachine>
```

3.2 Model Transformations: From Graphical State-machines to Ada Code

As already commented in section 1, model transformation is one of the key concepts of the MDE process, being one of the most powerful tools for designers. Actually, the OMG is about to finish the specification of the *MOF Query-View-Transformation* (QVT) [3], as the standard language for model transformations for their MDA approach.

Models can be transformed into other models or into any textual representation. *Model-to-Model* (M2M) transformations allow designers to refine abstract models (defined at the beginning of the design process) to obtain models that are closer to the final solution. These M2M transformations require defining a mapping between the corresponding meta-models, i.e. a transformation between the concepts represented in each meta-model. Once models are close enough to implementation, they can be transformed into a textual representation (e.g. code), using a *Model-to-Text* (M2T) transformation. Currently, it is possible to find some tools that make it possible to define both M2M and M2T transformations. Probably in the future, these tools will provide some interactive environment that will allow designers to guide the transformation process.

Among the M2T transformation tools currently available, we have chosen *MOFScript*[2] to generate Ada code from the state-machine models previously built using the GMF tool presented in section 3.1. MOFScript enables the description of M2T transformations both in a declarative and in an imperative way. An excerpt of the MOFScript code implemented to obtain Ada code from any of the state-machine models previously defined, is shown below.

Excerpt of the MOFScript code implemented for transforming models into Ada code:

```
texttransformation SM_Model2Ada1 (in MetaModel:"StateMachineTool")
{
  MetaModel.StateMachine :: main ( ){
    ...
    // Writing StateMachine.adb
    file sm_adb (self.name + ".adb")
    var initialTransitions:List=self.transitions->select (
          t: MetaModel.Transition |
          t.source.oclIsTypeOf(MetaModel.PseudoState))
    var initial:MetaModel.Transition=initialTransitions.first()
    sm_adb.println("with Ada.Text_Io; use Ada.Text_Io;");
    sm_adb.print  ("with " + self.name+"_Code" + "; use ");
    sm_adb.println(self.name + "_Code" + ";");
    sm_adb.println("package body " + self.name + " is")
    sm_adb.print("\t Current_State : T_State := " )
    sm_adb.println (initial._getFeature("target").stateName+";")
    writeFunction("Get_Current_State","T_State","return Current_State;")
    transitionNames -> forEach ( tn:String ) { writeProcedure ( tn ) }
    sm_adb.println ("end " + self.name + ";")
    ...
  } // main
} // texttransformation
```

As stated in [9], state-machines can be implemented using (1) big `if..else` (or `case`) like blocks, that define state changes depending on the transition being fired;

[2] `http://www.eclipse.org/gmt/mofscript/`

(2) a look-up table that relates transitions and states; or (3) applying the State pattern. Each of these possible ways of coding the logic of a state-machine can be implemented by simply defining different M2T transformations. Due to limitations of space, only one of the possible transformations of type (1) will be presented. This transformation produces two Ada packages (both specification and body) and an animator for the generated state-machine. One package contains the logic of the transitions between the different states, while the other defines all the activities present in the model (*onEntry, do, onExit* and *fire*). In the first package, each Transition is associated to an Ada `procedure`, and each State is part of an enumerated type. An excerpt of the Ada code resulting from applying this M2T transformation to the model depicted in Fig. 3 is shown below. The package *SMExample* contains the logic of transition between states, while package *SMExample_Code* (not shown here) defines the activities. This schema follows a separation of concerns approach and avoids the unnoticed modification of the logic of the state-machine while the user was filling the code of the different activities.

Excerpt of the Ada code generated from the model shown in Fig. 3:

```
with Ada.Text_Io; use Ada.Text_Io;
with SMExample_Code;
package body SMExample is
   -- -------------------------------------
   Current_State : T_State := Init;
   -- -------------------------------------
   function Get_Current_State return T_State is
   begin
      return Current_State;
   end Get_Current_State;
   -- -------------------------------------
   procedure Start is
   begin
      case Current_State is
           when Init =>
                SMExample_Code.Init_onExit;
                SMExample_Code.Start_fire;
                SMExample_Code.Idle_onEntry;
                Current_State := Idle;
                SMExample_Code.Idle_do;
           when others => null;
      end case;
   end Start;
   -- -------------------------------------
   ...
   -- -------------------------------------
   procedure Check is
   begin
      case Current_State is
           when Idle => SMExample_Code.Idle_do;
           when others => null;
      end case;
   end Check;
   -- -------------------------------------
end SMExample;
```

Lastly, the `null` statement in the `when others` line defines the reaction of the state-machine when the transition just triggered is not defined for the current state. This is just another way of handling semantic variation points, as stated in UML and [8].

Of course there can be different M2T transformations to support different policies for this semantic variation point, e.g. raise an exception. Another possible implementation for type (1) transformation could be to define both transitions and states as enumerated types, while using a unique and big `procedure` to specify the behaviour of the state-machine; again, it is also possible to define yet another M2T transformation to do this, which shows the potential benefits of developing software using a MDE approach.

4 Related Work

As already stated in the introduction, the purpose of this paper is to present a simple but complete example, based on state-machines, to demonstrate the benefits of applying a Model-Driven approach to software development. Although there are some tools that support the MDE approach (actually not many), we have chosen *Eclipse* as it is an open source project. Some of the alternative MDE environments we also considered were *MetaEdit+*[3] and *Microsoft Visual Studio with Domain-Specific Language Tools*[4], both of them available as commercial tools.

MetaEdit+ enables domain specific meta-model definition, graphical model specification and template-based model-to-code transformation. However, MetaEdit+ lacks two of the cornerstones of MDE, namely: (1) the underlying meta-meta-model is not available (Eclipse uses the MOF standard) and (2) it is not possible to define model-to-model transformations. Without a meta-meta-model it is not possible to clearly define and manipulate the meta-model elements, while without model-to-model transformations it becomes impossible to define different modelling abstraction levels.

On the other hand, the Visual Studio DSL Tools are the Microsoft answer to the OMG MDA initiative and they tightly integrated with other Microsoft and .NET tools. According to [10], "software factories use custom collections of DSLs to provide sets of abstractions customised to meet the needs of specific families of systems, instead of taking a generic, one-size-fits-all approach".

Although the state-machine graphical modelling tool and the model-to-Ada code transformation have only been developed as an example of the benefits of the MDE approach, it is worth to compare these tools with other well-known state-machine CASE tools. State-machines have been used in software system design since the early 1970s, being particularly useful in the embedded system domain. As a consequence, many different tools are currently available for describing and implementing state-machines in different programming languages. Among them, probably one of the most widely used is STATEMATE [11]. Later on, the UML adopted state-machines for describing the behaviour of the elements being modelled. Thus, new visual and UML-compliant tools appeared in the marketplace (e.g. Rational Rose, Raphsody or Poseidon, among many others), allowing designers to model and implement this artefacts. As the scope of these tools is wider than just generating code for state-machines, they commonly produce complex and cumbersome code making it difficult to extract the state-machine code. In this sense, the main advantage of the MDE approach is that developers can

[3] `http://www.metacase.com`

[4] `http://msdn2.microsoft.com/en-us/vstudio/aa718368.apx`

decide the abstraction level and the scope of their applications and the way models are transformed into code, giving them the full control of the development process.

5 Conclusions and Future Research Lines

Currently, Model-Driven Engineering (MDE) is considered one of the most promising approaches for software development. MDE merges new and matured technologies and is supported by an increasingly growing academic and commercial community.

In this paper we have presented a simple but complete example that demonstrates the great benefits of applying a MDE approach, both for modelling and implementing software systems. State-machines have been chosen as our example domain since these artefacts are quite simple and well known in the real-time community.

In order to allow designers to describe state-machines, we have defined a modelling language (meta-model) which is a simplified version of the one included in the UML 2.0. From this meta-model we have implemented a graphical modelling tool aimed at easing the specification and validation of state-machine models. These models can then be used to automatically generate Ada code using a model-to-text transformation also implemented as part of this work. All the specifications and applications presented in this paper have been implemented using some of the currently available MDE tools offered by Eclipse: such as EMF, GMF, EMFT-OCL and MOFScript, among others.

Although we have outlined different possible implementations of state-machines in Ada, only one of them has been addressed in this paper by means of a model-to-Ada code transformation. This transformation generates, among other Ada files, a simulation program that allows users to test the correctness and performance of the generated code.

Currently we are working on the definition of new model-to-text transformations in order to test the performance of other Ada state-machine implementations. We are also working in the definition of other model-to-code transformations to test different language implementations, in particular Java and VHDL. In the future we plan to extend the state-machine meta-model (and thus the graphical modelling tool) in order to include new domain concepts such as orthogonal regions and new pseudo-state kinds (i.e. join, fork, etc.).

Despite of the promising results shown by the MDE approach, it is still at a very early stage and there is a lot of research to be done before it can exhibit all its potential. Nevertheless, the increasingly growing interest of the software engineering community in the MDE approach envisages very good results in the coming years. Probably, one of the most promising features of this approach, in particular for the safety-critical real-time systems community, is in the field of automatic software V&V and certification. Some related research areas in this field include, among others: formal model transformations, automatic test generation, and robust and efficient code generation.

References

[1] Model Driven Architecture Guide Version v1.0.1, omg/2003-06- 01. Object Management Group (OMG) (2003), Available online:
`http://www.omg.org/docs/omg/03-06-01.pdf`

[2] Meta-Object Facility (MOF) Specification v2.0, ptc/04-10-15. Object Management Group (OMG) (2004), Available online:
http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF
[3] Meta-Object Facility (MOF) v2.0 Query/View/Transformation Specification, ptc/05-11-01. Object Management Group (OMG) (2005), Available online: http://www.omg.org/technology/documents/modeling spec catalog.htm#QVT
[4] Object Constraint Language (OCL) Specification v2.0, formal/06-05- 01. Object Management Group (OMG) (2006), Available online:
http://www.omg.org/technology/documents/modeling-spec- catalog.htm#OCL
[5] Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S.: Developing applications using model-driven design environments. IEEE Computer, vol. 39(2), IEEE Computer Society (2006) ISSN 0018-9162. doi: 10.1109/ MC.2006.54
[6] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Eclipse series. Addison-Wesley Professional, Reading (2003) ISBN 0131425420
[7] Bézivin, J.: On the unification power of models. Software and Systems Modeling 4(2), 171–188 (2005) doi: 10.1007/s10270-005-0079-0
[8] Chauvel, F., Jézéquel, J.M.: Code Generation from UML Models with Semantic Variation Points. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 54–68. Springer, Heidelberg (2005) ISBN 3-540-29010-9. ISSN 0302-9743, 2005. doi: 10.1007/11557432-5
[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, Reading (1995) ISBN 0201633612
[10] Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, 1st edn. Wiley, Chichester (2004) ISBN 0471202843
[11] Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. In: ACM Trans. Softw. Eng. Methodol., vol. 5(4), ACM Press, New York (1996), ISSN 1049-331X doi:10.1145/235321.235322
[12] Kent, S.: Model Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002) ISBN 3-540-43703-7. ISSN 0302-9743
[13] Schmidt, D.C.: Model-Driven Engineering. In: IEEE Computer, vol. 39(2), IEEE Computer Society Press, Los Alamitos (2006) ISSN 0018-9162 doi:10.1109/MC.2006.58