A model-driven engineering approach for rapid real-time system simulations

Alain Plantec, Frank Singhoff, Mickaël Kerbœuf
Université Européenne de Bretagne
Université de Brest, France
Laboratoire d'Informatique des Systèmes Complexes (LISyC)
{plantec,singhoff,kerboeuf}@univ-brest.fr

Abstract

Because it encourages the incremental development of software and the reuse of components by abstracting away platform dependent details, model-driven engineering is an intuitive and sensible way to conceive large software out of existing application components and libraries. In practice, however, just a few practical tools make it possible to generate partially automatically but efficiently large scale industrial applications.

We introduce in this article a meta-modeling tool called Platypus. It enables to specify very quickly within a homogenous framework a model and the meta-model with witch it complies. In this generic framework, some generated components can dynamically enrich the framework itself in order to incrementally adapt it to a specific domain.

The benefits of this tool are illustrated by a concrete and practical example: the adaptation of Cheddar, a simulation tool designed to real-time software system analysis.

Introduction

This article deals with the use if a model-driven engineering tool in the context of real-time system verifications.

Many meta-modeling tools are now available. The most known are *MetaEdit+* [15] and the *EMF* framework [5]. They provide a meta-modeling language and a set of tools allowing meta-modeling and domain editors implementation. Specific source code or documentation generators can be implemented using a dedicated language. Meta-models and complying models are usually edited using a graphical user interface with boxes and lines. They are multi-language based. Classically, these languages are designed for meta-modeling, for source code generating and optionally, for meta-constraints expressing.

Platypus [17, 19] is a *STEP* based environment implemented within the free Smalltalk system Squeak [21]. The distinctive feature of *Platypus* is the use of a unique language to specify meta-models as also as domain constraints, translation rules and domain complying models. This language is *EXPRESS*, a data modeling language.

Cheddar is a library designed for the performance analysis of real-time systems. Within *Cheddar*, a real-time system is modeled as a set of software and hardware components such as tasks, processors, schedulers, or buffers. These components can be specified with the domain specific language of *Cheddar*, or with AADL [20]. *Cheddar* provides a set of real-time schedulers and their analysis tools implemented in Ada. Schedulers currently implemented in *Cheddar* are mostly used in real-time systems. *Cheddar* can be used to perform performance analysis of many different types of real-time systems. However, it exists a need to extend these *Cheddar* analysis tools with user-defined schedulers or task models. Extending *Cheddar* with new schedulers or new task models requires that the user well understands the design of *Cheddar*. Furthermore, specifying a new scheduler or a new task model may be difficult without an environment especially designed to easily write and test the scheduler source code. In order to ease the specification of new schedulers, *Cheddar* provides a specific programming language. The model of a scheduler described and tested with the *Cheddar* programming environment is interpreted. Thus, the designer can easily experiment his scheduler models.

Some case studies showed that the interpreter lacks of efficiency on large scheduling simulations. Regarding this problem of performance, an optimal solution is to re-implement this kind of scheduler with the *Cheddar* library and integrate it as a built-in one within *Cheddar*.

However, this solution is expensive and error prone because of the complexity of the domain. We propose to translate automatically *Cheddar* schedulers into their equivalent Ada programs with the help of *Platypus*.

In this article, we present *Platypus* and the associated meta-modeling methods used in order to implement a part of *Cheddar*. This experiment shows how real-time system

performance analysis tools can be automatically produced with model-driven engineering tools. These tools help designers to verify the design of their systems at an early stage.

Different experiments have shown how model-driven engineering method and tools can help system prototyping. For example, ASSERT [8] has proposed a process based on AADL [20] for such a purpose.

The main other software engineering tools that are able to model and generate real-time system software are STOOD (Ellidiss Technologies), Artisan Studio, UML STP (AONIX), Rhapsody (Telelogic) or Rational Rose [4, 6, 11, 12]. Most of them are built with a fixed meta-model which enforces the domain specific features of real-time systems.

This article is organized as follow. The first section describes *Platypus* tool. The second section gives an outline of the *Cheddar* library. Then, in the third section, we explain how *Platypus* is used to automatically produce *Cheddar* software components. Finally, conclusion and ongoing-works are presented in the fourth section.

1 Platypus, a STEP/EXPRESS based meta-environment

Platypus is a meta-environment fully integrated inside *Squeak* [21], a free *Smalltalk* system. *Platypus* allows meta-model specification, integrity and translation rules definition. Meta-models are instantiated from user-defined models. Given a particular model, integrity and translation rules can be interpreted.

Platypus provides only textual meta-modeling and modeling facilities. *Platypus* benefits from the STEP [9] standard for models and meta-models specification and implementation. *STEP* is an ISO standard which was developed to easily share product informations by specifying sufficient semantic for data and their usage. Within *STEP*, models are specified with the data oriented modeling language *EXPRESS* [10].

Platypus makes use only of *EXPRESS* to build models and meta-modeling, and to specify constraints and code generators. In *Platypus*, models and meta-models consist in sets of *EXPRESS* schemas. A schema is a root element of an *EXPRESS* specification. A schema contains primary modeling elements which are constants, types, entities, procedures, functions and global rules. Entities are used in order to specify domain concepts. An entity contains a list of attributes that provide buckets to store meta-data while local constraints are used to ensure meta-data soundness.

Platypus is primarily a *STEP* environment that involves an editor for *EXPRESS*, a parser, an interpreter and two built-in Smalltalk generators that allow *EXPRESS* models to be mapped in Squeak [21] and VisualWorks [22].

1.1 Platypus and the STEP technical space

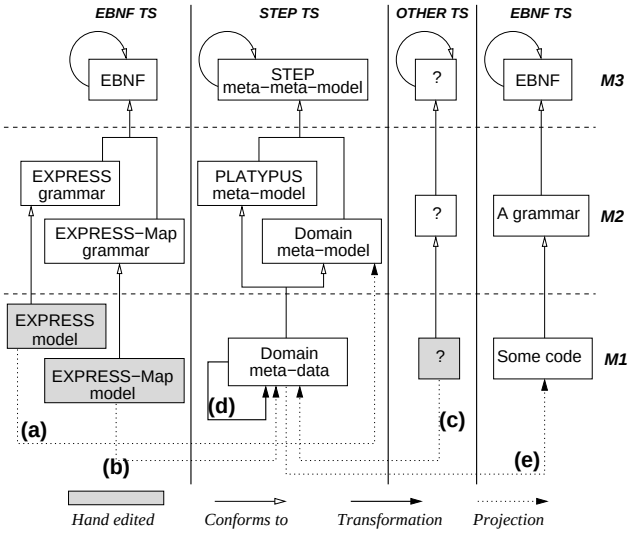


Figure 1. Model transformation and projection with Platypus

Figure 1 depicts a *Platypus* architecture using the concept of *technical space*. A *technical space* (TS) is defined as a set of models and the tools that can operate on these models [13]. In figure 1, technical spaces are presented vertically. Each one is made of three rows. Each row corresponds to a level of the MDA four levels architecture [16]. A generative operation across several TS is called a *projection*. A generative operation inside the same TS is called a *transformation*.

The main TS is the *STEP* TS in which *Platypus* is built. *Platypus* is implemented around a fixed meta-meta-model that is mainly a *STEP* core meta-model (M3 level) and around the *Platypus* meta-model (M2 level). The *Platypus* meta-model complies with the *STEP* meta-meta-model and is fixed for a particular version of *Platypus*.

As a meta-environment, *Platypus* provides two main functionalities. The first one allows meta-modeling, which consists in editing meta-models. The second one allows the implementation of meta-models which at least consists in meta-data instanciation, browsing, checking and transformation.

Platypus can be specialized in order to handle a specific domain. Such a specialization is called a domain specific environment. The next sections explain how such domain specific environments are specified and used.

1.2 Specifying a domain specific environment

In order to specialize *Platypus*, a designer only has to specify a meta-model which describes the concepts of the domain he would like to handle. *EXPRESS* is used as a domain modeling language. The meta-model is edited in the EBNF TS (EBNF stands for *Extended Backus-Naur Form*) and is projected by *Platypus* in the *STEP* TS at the M2 level (see (a) in figure 1).

The *Platypus* meta-model is itself an EXPRESS model. Since it mainly consists in an EXPRESS language meta-model, a user-defined meta-model can reuse and specialize it. In this way, a user can specify a domain specific specialization of the EXPRESS language and thus, make *Platypus* a domain specific environment.

1.3 Using a domain specific environment

Using a domain specific environment consists in providing meta data, and in checking them or using them in order to produce some realization. These two points are explained below.

1.3.1 Providing meta data

Meta data building can be either made externally, by a tool implemented outside *Platypus* or made internally with the help of the mapping feature of *Platypus*.

External meta data producing is represented in figure 1 by the projection from an *a priori* unknown *Other TS* to the *STEP* TS (see (c) in figure 1). A specific tool implementation can be a difficult and a costly task. However, if the meta-model has been specified as a specialization of the *Platypus* meta-model, this implementation is not mandatory because internal meta data producing can be used with the help of an *EXPRESS-map* model.

EXPRESS-map is a *Platypus* specific extension of *EXPRESS* that allows the definition of *conform-to* relations between an EXPRESS model and a meta-model. Then, in order to use a domain specific meta model, the user has to provide a domain model written in standard *EXPRESS* and a mapping model written in *EXPRESS-map*. Figure 2 depicts an example of an explicit mapping.

The benefit is that *Platypus* is able to automatically implement a projection from the EBNF TS to the *STEP* TS (see (b) in figure 1). Such a projection builds meta-data related to a domain specific model. This feature allows very rapid implementation of domain models since the user has only to deal with domain meta-models and domain models specification.

```
ENTITY Record SUBTYPE OF (entity_definition);
END_ENTITY;
```

```
ENTITY Buffer;
END_ENTITY;
```

```
MAP Buffer TO Record;
END_MAP;
```

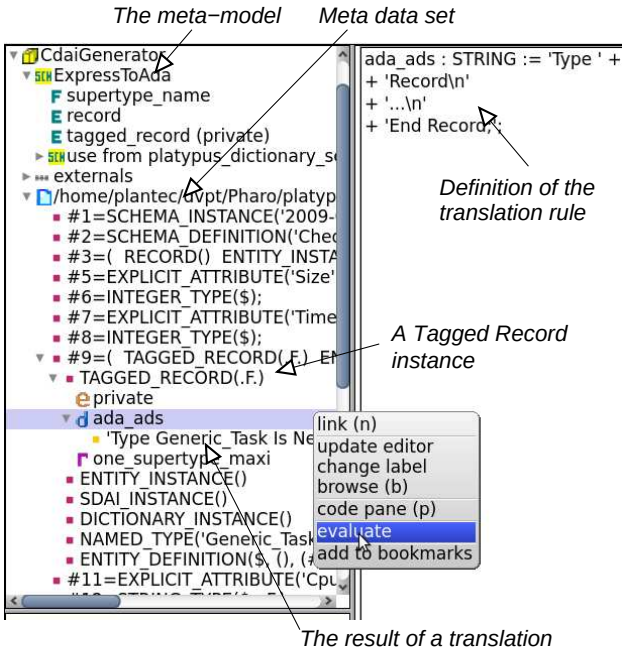
The upper frame shows the entity *Record* from a user-defined meta-model which is specializing the *Platypus* meta-model for a domain specific purpose. The middle frame shows the entity *Buffer* from a user-defined model and the lower frame shows a mapping declaration.

Figure 2. Explicit mapping

1.3.2 Using meta data

After a model has been provided, constraints and translation rules defined within the meta-model allows meta data checking, transformations and projections.

Typically, a projection from the *STEP* TS to the EBNF TS is implemented as a derived attribute which result is a string (see (e) in figure 1). A transformation within the *STEP* TS is implemented as a derived attribute which result is a new meta-data (see (d) in figure 1).



Users can browse the meta data, select a derived attribute (e.g. an *ada_ads* attribute of a *Tagged_Record* instance) and evaluate it. Then, the corresponding result is stored as a child of the selected attribute. The same kind of interaction is possible for the checking of constraints.

Figure 3. Translation from *Platypus* tool

Constraints and translation rules are interpreted by a generic *Platypus* component called a repository. Using of

a repository is made through the *Platypus* model browser with which all elements of a model can be visualized. The checking of constraints and the evaluation of derived attributes are available from the model browser itself. Figure 3 shows a snapshot of *Platypus*. This snapshot depicts how one can evaluate a translation rule from the user interface.

Regarding the example of *Cheddar*, we now explain the analysis tool that we expect to generate with the meta-environment *Platypus*. First, we briefly present *Cheddar* architecture, and then, the applied model-driven engineering process.

2 Cheddar architecture outline

As shown by figure 4, *Cheddar* is made of six main components and the overall architecture is made of two layers.

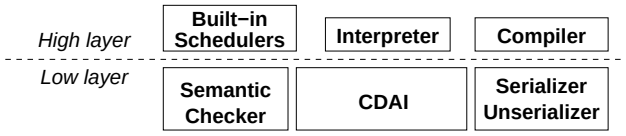


Figure 4. Major Components of *Cheddar*

2.1 The low layer

The low layer is built around a repository for data and meta-data storage. Data and meta-data accessing to and from the repository are all implemented by the *Cheddar* Data Acces Interface (*CDAI*). The *CDAI* is a central component that is used by all other *Cheddar* components.

The lower layer implements some additional data specific components such as a data checker component and a data exchange component which is responsible for the writing and the reading of XML data files.

2.2 The high layer

This layer allows real-time systems simulation at two levels:

1. *Cheddar* natively implements several well known scheduling algorithms. These schedulers are hand-written. Ada components implementing these schedulers are called ”built-in schedulers”.

2. The compiler and the interpreter are respectively responsible for the compiling and the running of user-defined schedulers programmed with the *Cheddar* language.

3 Cheddar engineering with Platypus

The use of *Platypus* is twofold. First, the *Cheddar* low layer, which is dedicated to data management, is generated with *Platypus*. This layer directly depends on manipulated data types and constraints. It is very generic in nature and its components are classically automatically generated from the specification of related data types.

The goal of the second use is to make it possible the translation of *Cheddar* programs to Ada components. Then, a user-defined scheduler programmed with the *Cheddar* language can be integrated within the *Cheddar* library as any built-in scheduler Ada components.

In the sequel, we focus on implementation of source code generators with *Platypus*. First, section 3.1 explains with details the CDAI source code generator implementation. Since both CDAI source code generator and scheduler source code generator has numerous similarities, section 3.2 just outlines the source code generator devoted to user-defined schedulers.

3.1 The CDAI generator

Cheddar deals with a clearly defined set of primary data types which are tasks, processors, schedulers, buffers, ... This data model (called the *Cheddar* data model in the sequel) allows users to specify the real-time system architectures that they expect to analyze. We decided to use *Platypus* in order to generate a clean source code according to well defined coding rules [18]. The main goal is to provide a clear data accessing interface (the CDAI) implementing a standardized data access protocol. Figure 5 depicts the translation schema.

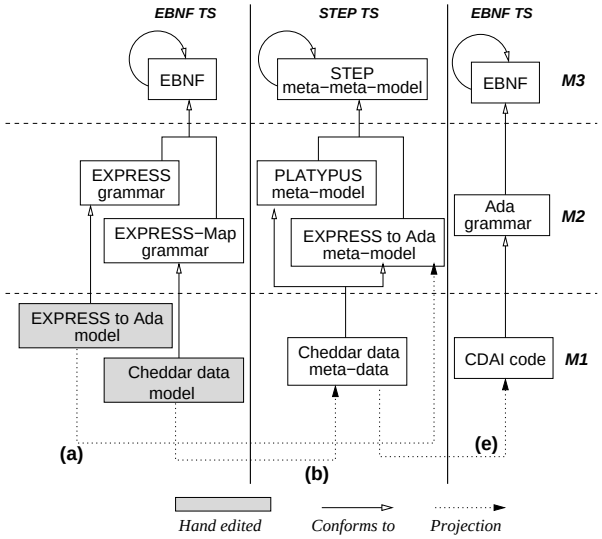


Figure 5. Generation of the CDAI

Hand edited *EXPRESS* models are the *ExpressToAda* meta-model and the *Cheddar* data model. The *ExpressToAda* meta-model specifies the translation rules whereas the *Cheddar* data model specifies the primary data types manipulated by *Cheddar*. *ExpressToAda* meta-model is projected to the *STEP* TS at M2 level and the *Cheddar* data model is projected at the M1 level of the *STEP* TS (see (a) and (b) in figure 5) with the help of the mapping feature.

3.1.1 Meta-modeling: specification of *ExpressToAda*

It consists in the specification of target concepts and the associated translation rules. Translation rules allow the projection of *EXPRESS* constructs to target Ada language constructs and to CDAI related Ada sub-programs. The meta-model is specified as a specialization of the *Platypus meta-model*: each Ada construct, mainly *Package*, *Record* and *Tagged record*, is defined as a subtype of a concept from the *Platypus* meta-model, mainly, *Schema* and *Entity* definitions.

```
SCHEMA platypus_dictionary_schema; ...
  ENTITY named_type SUBTYPE OF (dictionary_instance);
    name : string;
    where_rules : LIST OF where_rule; ...
  END_ENTITY;

  ENTITY entity_definition SUBTYPE OF (named_type);
    supertypes : LIST OF entity_definition;
    attributes : LIST OF attribute; ...
  END_ENTITY; ...
END_SCHEMA;

SCHEMA ExpressToAda;
  USE FROM platypus_dictionary_schema;

  ENTITY record SUBTYPE OF (entity_definition);
  DERIVE
    ada_ads : STRING := 'Type ' + SELF.name + ' is\n'
    + 'Record\n' + '...\n' + 'End Record; ';
  WHERE
    no_supertype : SIZEOF ( supertypes ) = 0;
  END_ENTITY;

  ENTITY tagged_record SUBTYPE OF ( entity_definition );
    private : BOOLEAN;
  DERIVE
    ada_ads : STRING := 'Type ' + SELF.name + ' is New '
    + supertype_name ( SELF ) + '\n'
    + 'With Record\n' + '...\n' + 'End Record; ';
  WHERE
    one_supertype_maxi : SIZEOF ( supertypes ) <= 1;
  END_ENTITY;

  FUNCTION supertype_name(tg : tagged_record): STRING;
    IF ( SIZEOF ( tg.supertypes ) = 0 ) THEN
      RETURN ( 'Ada.Finalization.Controlled' );
    ELSE
      RETURN ( tg.supertypes [ 1 ].ref.name );
    END_IF;
  END_FUNCTION;
END_SCHEMA;
```

Figure 6. The *ExpressToAda* meta-model

A very simplified version of *ExpressToAda* is shown in figure 6.

platypus_dictionary_schema is a part of the reused *Platypus* meta-model. It is read-only because the used version of *Platypus* engine depends on it. It owns *entity_definition* meta-entity that specifies what an *EXPRESS* concept is. *entity_definition* inherits from *named_type.* An entity has a name (*name* attribute), a list of local constraints (*where_rules* attribute), a list of supertypes (*supertypes* attribute) and a list of attributes (*attributes* attribute).

Figure 6 presents *Record* and *Tagged record* Ada concepts specification. A tagged record is an Ada construct which is equivalent to a Java class. *tagged_record* private attribute is added because the concept of privacy which is useable in Ada isn't available in *EXPRESS*. Ada source code is computed by the derived attribute *ada_ads*; Each concept definition can own constraints. Such a constraint is useful statically as well as dynamically in order to, respectively, provide a rich documentation of the meta-model and to allow the validation of meta data before any projection is computed. Constraints are defined in order to ensure that projections can be computed.

3.1.2 Data modeling: the *Cheddar* data model

The modeling activity consists in *Cheddar* primary data types specification. Figure 7 shows a very simplified version of *Cheddar* data model with three data types which are *Buffer*, *Generic_Task* and one of its specialization, *Aperiodic_Task*. *CheddarData* is a standard *EXPRESS* model. It can be used as input for other *EXPRESS* related tools. As an example, an external tool can use this model in order to produce some source code or some other model. In other terms, *CheddarData* can be considered as a pivot representation of *Cheddar* around which other tools can be articulated. As an example, *CheddarData* can serve as a domain and *Cheddar* reference model to perform analysis of Marte/UML real-time system models [14].

```
SCHEMA CheddarData;

  ENTITY Buffer;
    Size : INTEGER;
    Time : INTEGER;
  END_ENTITY;

  ENTITY Generic_Task;
    Cpu_Name : STRING;
    Capacity : INTEGER;
    Deadline : INTEGER;
    Start_Time : INTEGER;
  END_ENTITY;

  ENTITY Aperiodic_Task
    SUBTYPE OF ( Generic_Task );
  END_ENTITY;
END_SCHEMA;
```

Figure 7. The *CheddarData* model

3.1.3 EXPRESS-map modeling

```
SCHEMA CheddarData_To_ExpressToAda_Mapping;
  META FROM ExpressToAda;
  USE FROM CheddarData;

  MAP Buffer TO record ( );
  END_MAP;

  MAP TO tagged_record ( false );
    Generic_Task;
    Aperiodic_Task;
  END_MAP;
END_SCHEMA;
```

Figure 8. The mapping schema for *CheddarData*

The two previous sections have described both the meta-model and the model designed to automatically produce the real-time systems simulation tool components. We now explain how to specify relationships between these meta-model and model.

Figure 8 shows a mapping model for the CDAI generator example. A mapping model is made of two parts. The first part is the declaration of used meta-models and models. Used meta-models are declared with the *META FROM* expression and used models are declared with the standard *EXPRESS* expression *USE FROM*. The second part is made of the declaration of the *conform to* relations for the model elements. In this example:

- *Buffer* is declared as conform to a *Record*,
- *Generic_Task* and *Aperiodic_Task* are declared as conform to a *Tagged_Record*.

Then, the projection from the EBNF TS to the *STEP* TS can be driven according to these mapping rules: it builds an instance of the *Record* meta entity from the *Buffer* entity and two instances of the *Tagged_Record* meta entity from both *Generic_Task* and *Aperiodic_Task* entities.

As a consequence, the CDAI generator considers that a *Buffer* is checked or translated to Ada components following respectively the constraints and the translation rules declared by the *Record* specification of the *ExpressToAda* meta-model. The same process is applied to *Generic_Task* and *Aperiodic_Task* but with the *Tagged_Record* meta entity.

3.2 The scheduler generator

User-defined schedulers can be programmed with the *Cheddar* language. The *Cheddar* language is a small domain specific language. A *Cheddar* program modeling a new scheduler is organized as a set of timed automata such as those proposed by UPPAAL [7, 1, 2].

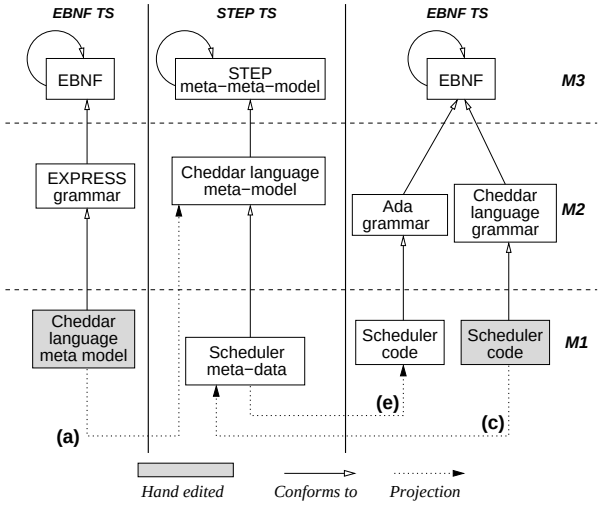


Figure 9. Source code generation of schedulers

Figure 9 depicts the translation schema. The *Cheddar* language grammar is an EBNF grammar. The meta-model for this language is specified with *EXPRESS* and is projected to the *STEP* TS at M2 level (see (a) in figure 9). This meta-model specifies the *Cheddar* language constructs (automaton, expression and statement types). The using of this meta-model is twofold:

- This meta-model is used as input to the CDAI generator, then, within *Cheddar*, a part of the CDAI is dedicated to the management of *Cheddar* program meta data.
- Meta entities of the *Cheddar* language meta-model are defined with their own translation rules which specify how to produce an Ada scheduler component.

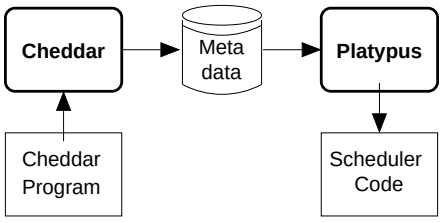


Figure 10. Meta data exchange between *Cheddar* **and** *Platypus* **for a scheduler source code generation**

Figure 10 depicts the meta data flow. For the generation of a scheduler component, external meta data producing is implemented. *Cheddar* is itself responsible for their producing. Thank to the CDAI, from a given program, *Cheddar* is

able to generate an exchange file. This file contains the meta data corresponding to the program. These meta data comply with the *Cheddar* language meta-model. Then, *Platypus* is able to read and use these meta data for the generation of a Ada scheduler component: *Platypus* instanciates the *Cheddar* language meta-model and evaluates its translation rules (respectively (c) and (e) in figure 9).

4 Conclusion

This article has presented *Platypus* and the associated meta-modeling methods used in order to implement a part of *Cheddar*, a real-time system analysis tool. These tools aim at helping designers to verify the design of their systems at an early stage. This experiment shows how performance analysis tools can be automatically produced with a model-driven engineering tool.

So far, two code generators were proposed. A first one is responsible for the generation of the data management layer of *Cheddar*. It is fully implemented and generates all Ada components related to model and meta-model data required for performance analysis of a real-time system. The implementation of the second one is in progress. It will be able to automatically generate Ada packages from the user-defined schedulers expressed with the domain specific language of *Cheddar*. The scheduler generator will give to the users the possibility to produce new versions of *Cheddar* implementing their own schedulers.

Some large scheduler models already exist. For example, Airbus Industries has developed a model of a flight simulator architecture that is composed of several scheduler *Cheddar* programs [3]. Simulations are operationnal with these models but require a large amount of memory and computing resources. There is a need to speed-up these simulations and source code generators presented in this article could be useful in this context. It is planned to evaluate source code generators by experiments with large scale scheduler models. We expect to perform these experiments with scheduler models proposed by Airbus.

References

[1] R. Alur and D. L. Dill. Automata for modeling real time systems. Proc. of Int. Colloquium on Algorithms, Languages and Programming, Vol 443, LNCS, pages 322–335, 1990.

[2] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. Technical Report Updated the 17th November 2004, Department of Computer Science, Aalbord University, Denmark, 2004.

[3] M. Castor, J. Casteres, and F. Gasmi. Modélisation et simulation de l'Architecture des simulateurs avion pour la mesure de performance. Rapport technique Airbus, September 2007.

[4] P. Dissaux. AADL Model transformations. In the DAta Systems in Aerospace conference (DASIA 2005), Edinbugh, July 2005.

[5] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf.

[6] M. Hause. Artisan Studio : support for Model Driven Architecture (MDA). *White paper of Artisan Software Tools*, 2002.

[7] J. E. Hopcroft and J. D. Ullman. Introduction of Automata Theory, Languages and Computation. Addison-Wesley editor, 2001.

[8] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computing Systems (TECS) , ACM Press, New York, USA*, 7(4):42:2–42:25, July 2008.

[9] ISO 10303-1. *Part 1: Overview and fundamental principles*, 1994.

[10] ISO 10303-11. *Part 11: edition 2, EXPRESS Language Reference Manual*, 2004.

[11] J. Rumbaugh and I. Jacobson and G. Booch. The Unified Modeling Language - Reference Manual. *Addison-Wesley*, 1999.

[12] M. Kersten, J. Matthes, C. F. Manga, S. Zipser, and H. B. Zeller. Customizing UML for the development of distributed reactive systems and code generation to Ada 95. *Ada User Journal*, 23(6), 1999.

[13] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, NY, USA, 2006. ACM.

[14] E. Maes. Validation de systèmes temps-réel et embarqué à partir d'un modèle MARTE. Thales RT, Journée Ada-France 2007, Brest, décembre 2007.

[15] MetaEdit+ Technical Summary. http://www.metacase.com/papers/index.html.

[16] OMG. Model Driven Architecture. *http://www.omg.org/mda*, 2003.

[17] A. Plantec and V. Ribaud. Platypus : A step-based integration framework. In *14th Interdisciplinary Information Management Talks (IDIMT-2006)*, Sept. 2006.

[18] A. Plantec and F. Singhoff. Refactoring of an Ada 95 Library with a Meta CASE Tool. *ACM SIGAda Ada Letters, ACM Press, New York, USA*, 26(3):61–70, November 2006.

[19] Platypus web site. http://cassoulet.univ-brest.fr/mme.

[20] SAE. Architecture Analysis and Design Language (AADL) AS 5506. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 1.0, November 2004.

[21] Squeak web site. http://squeak.org.

[22] VisualWorks web site. http://cincomsmalltalk.com/.