Refactoring of an Ada 95 Library with a Meta CASE Tool

Alain Plantec, Frank Singhoff
LISYC/EA 3883, University of Brest
20, av Le Gorgeu
CS 93837, 29238 Brest Cedex 3, France
{plantec,singhoff}@univ-brest.fr

ABSTRACT

This paper presents the refactoring work of *Cheddar*, a set of Ada packages which aims at providing performance analysis tools for concurrent real time applications. CASE tools can be used for such a purpose. However, we chose to use a meta CASE tool called *Platypus*. It seems that few studies exist concerning Ada and meta-modelization. Then, in this paper, we investigate how to use a meta CASE tool in order to automatically produce some parts of an Ada 95 object oriented software.

Keywords

Meta-modeling, Meta CASE, *STEP*, *EXPRESS*, Code generating, *Platypus*, *Cheddar*

General Terms

Design, Languages

Categories and Subject Descriptors

SOFTWARE ENGINEERING [**Design Tools and Techniques**]: Computer-aided software engineering (CASE)

1. INTRODUCTION

In [22, 23], we presented *Cheddar*, a set of Ada packages which aims at providing performance analysis of concurrent real time applications. With *Cheddar*, a real time application is modeled by a set of processors, shared resources, and tasks described by an AADL specification[7].

The development of this toolset started in 2002. Today, it includes most of classical scheduling simulation methods and classical scheduling feasibility tests in the case of dependent and independent tasks running on monoprocessor and distributed systems[5, 12].

We plan to strongly extend it in order to be able to analyze multiprocessor systems and hierarchical schedulers[20]. These new services will imply a large amount of modifications. Due to the toolset size (around 140000 lines) and due to the large amount of modifications we will have to do, we chose to perform a refactoring of this library with a CASE tool. From this refactoring work, the *Cheddar* team expects:

First, to strongly increase the *Cheddar* maintainability. Indeed, a large part of the *Cheddar* source code is composed of packages providing services to parse different application specification files, to check integrity constraints on data, to store these data into the simulation engine, and to present them on a machine-man interface. All these packages can be automatically produced from the *Cheddar* data model. By the past, doing changes on the *Cheddar* data model in order to implement new performance tools implied a huge amount of work on these packages. By using code generation, we expect to strongly reduce the cost of such future modifications on these packages.

Second, the use of CASE tools makes it possible to apply source code generation rules. These generation rules allow to tune the generated software according to user requirements. A good framework should be able to automatically take into account the user software configuration requirements. We expect to provide such a user configuration flexibility with specific source code generation rules.

Finally, we simply expect to improve the design and the reliability of the *Cheddar* framework.

Several Ada CASE tools are already available to the Ada community. We chose to use our meta CASE tool called *Platypus*[19] in order to investigate how meta-modelization can be applied to Ada. This paper shows how to use *Platypus* in order to automatically produce some parts of an Ada 95 object oriented software.

This paper is organized as follows. In section 2, we present what a meta CASE tool is. In section 3, we give an introduction to the *Platypus* meta-modeler and we describe the meta-modeling of *Cheddar* and Ada 95 with *Platypus*. Section 4 gives few details on the design of the *Cheddar* framework and the Ada packages we expect to generate. Finally, we conclude in section 5.

2. CASE AND META CASE TOOLS

CASE stands for **C**omputer-**A**ided **S**oftware **E**ngineering and is the use of software to assist in the analysis, the design, the implementation, the maintenance or the refactoring of software. A CASE tool is usually implemented according to a particular method or software implementation process. It automates the use of specific method modeling concepts or specific process steps and mainly provides modelization environments and code generators.

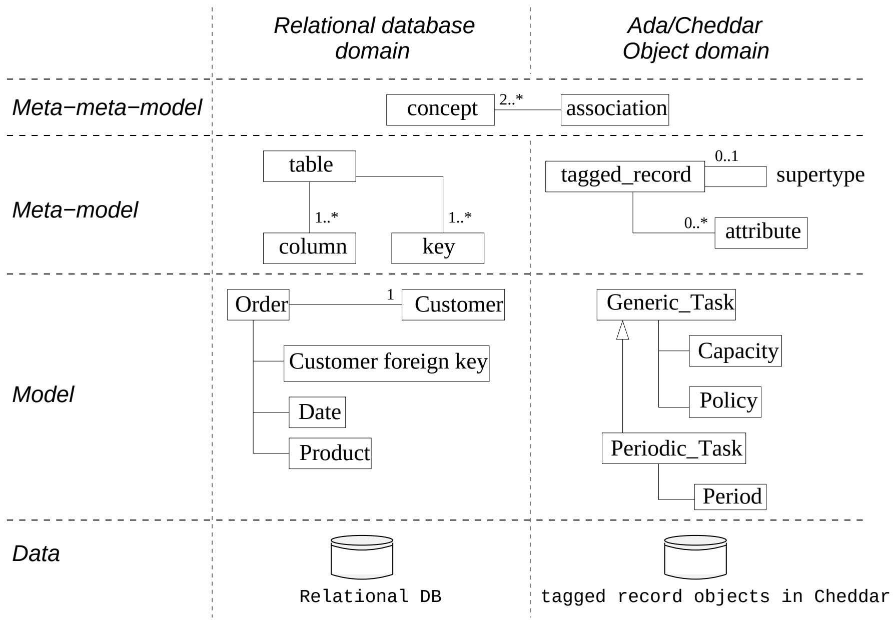


Figure 1: Meta-model, model and data

Classically, systems are built on a four layers architecture : meta-meta-models, meta-models, models and data[17]. The lowest layer is the application data layer. These data are instances of a model which is itself described by a language, usually called the meta-modele. Figure 1 shows two examples of meta-models, models and data. The meta-meta-model provides a minimal meta-modeling language.

The first example deals with relational data base systems. The meta-model describes the concepts of table, column and key. The meta-model is used to describe data base architectures. A particular data base architecture is described by a model. The data are the tuples of the data base. The second example is related to the work described in this paper: an Ada object oriented application. The set of concepts is composed of tagged record and attributes. The meta-model specifies such concepts of tagged record and attribute. The model of an application is the set of *Cheddar* tagged records (eg. $Periodic\_Task$) and the data are the tagged record instances.

Figure 2 shows the architecture of a CASE tool and a target application. A CASE tool is based on a fixed meta-model: method concepts are specifically implemented as builtin structures (eg. C++ classes or Ada tagged records). A CASE tool provides a mean to edit or elaborate models (eg. UML models). A model is internally handled as instances of builtin structures that are read by code generators from a repository in order to produce a realization. A database component is a typical realization. Often, CASE tools provide a framework made of generic libraries that are needed in order to compile or run target application. Comparing to a CASE tool, a meta CASE tool provides a way to edit meta-models and has a fixed meta-meta-model.

2.1 CASE tools available for Ada 95

Several Ada CASE tools such as STOOD (Ellidiss), Artisan Studio, UML STP (AONIX), Rhapsody (Telelogic) or

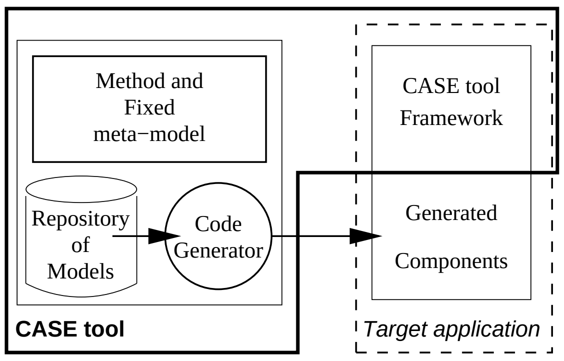


Figure 2: A CASE tool and a target application architecture

Rational Rose, are already available to the Ada community[2, 6, 10, 11]. In order to model and generate a domain specific software, two methods can be used by a CASE tool:

1. building the CASE tool and its generator on a meta-model which enforces the domain specific features ;

2. providing a meta-model which can be adapted to the specific domain.

Let see some examples of Ada CASE tools based on a fixed meta-model. In this section, we chose to present CASE tools designed for the specification and the generation of real time applications (which is the *Cheddar* domain).

Ellidiss technologies provides STOOD[2], a modeling tool for UML, AADL and HOOD. The STOOD meta-model is based on HRT-HOOD[1]. The HRT-HOOD model includes a set of features which enables the design of real time applications. The HOOD meta-model of STOOD allows a user to see the model of his application as an UML or an AADL design : STOOD is able to automatically translate a design towards UML, AADL or HOOD ; thanks to pre-defined mappings between the internal meta-model and the user level notations. Unlike most of the other CASE tools, STOOD does not use simple code template instanciation to generate the applicative code from the design model. Instead, code generation rules are formally defined using the LMP (Logical Model Processing) technology, and can be easily tuned to fit any specific project coding standard.

Let see some CASE tools which provide meta CASE facilities such as the one proposed by UML. Some UML CASE tools make it possible the definition of specific domain concepts with mechanisms such as stereotypes. A stereotype represents an UML usage distinction[10]. A particular stereotype is an unstandardized modeling concept that is tool dependent. It is expected that code generators will treat stereotyped UML designs in order to generate source code tuned to the specific domain[11].

2.2 Meta CASE tools

As described in [13], it exists many meta-modeling tools. The most known are *MetaEdit+*[14] and *Dome*[3]. They provide a minimal meta-modeling language that is general enough to specify a large amount of meta-modeling cases. They also provide a set of graphical tools allowing graphical meta-modeling and graphical domain editor definition. Specific code or documentation generators can be implemented using a dedicated language.

They provide a graphical way to specify meta-models and related model editor. They are multi-language based. Classically, these languages are for meta-modeling, code generating and optionally, meta-constraints expressing. A meta-modeler has to specify not only domain specific meta-models but also domain specific editors.

Usually, meta CASE tools are general purpose CASE tools. This is the case of the meta CASE tools presented above. However, it also exists some domain specific meta CASE tools : *TOPCASED*[4] is one of such an environment. As for *MetaEdit+* or *Dome*, *TOPCASED* can be use for the developement of any kind of applications, but it is also especially well suited for critical real time embedded systems design (one of the *Cheddar* domain application). The *TOPCASED* CASE tool is managed by the French Aeronautic and Space National Research Center. It is based on the Eclipse platform : meta-models are described with EMF[26].

3. THE PLATYPUS META CASE TOOL

Platypus[19] is a meta-environment fully integrated inside *Squeak*[24], a free *Smalltalk* system. *Platypus* allows meta-model specification, integrity and transformation rules definition. Meta-models are instantiated from user defined models and, given a particular model, integrity and transformation rules can be interpreted.

Platypus allows only textual meta-modeling and modeling facilities. *Platypus* benefits from the *STEP* standard for meta-models specification and implementation. As an *ISO* standard, *STEP*[8] is developed to facilitate product information sharing by specifying sufficient semantic content for data and their usage. Parts of *STEP* are intended to standardize conceptual structures of information either generic, or within a particular domain (e.g. mechanics). Standardized parts are expressed with a dedicated technology, mainly an object oriented modeling language called *EXPRESS*[9] and a data access interface. *EXPRESS* can be used as a meta-modeling language[18, 16].

Platypus is a mono-language tool: only *EXPRESS* is used for meta-modeling, constraints and code generator specification. In *Platypus*, a meta-model consists in a set of *EXPRESS* schemas that can be used to describe a language. The main components of the meta-model are types and entities. They are describing the language concepts. Entities contain a list of attributes that provide buckets to store meta-data while local constraints are used to ensure meta-data soundness.

A translation rule is defined within a meta-entity as a derived attribute: a named property which value is computed by the evaluation of an associated expression. A typical translation rule returns a string and can be parameterized with other meta-entities. The resulting string represents part of the target textual representation (eg. Ada source code, documentation, XML data).

Platypus meta-model can itself be reused for meta-modeling. This feature decreases domain specific environment implementation cost by allowing *Platypus* environment reusing not only for meta-modeling but also for modeling and code generator running.

3.1 Meta-modelization of Ada for Cheddar

Figure 3 shows a part of a simple Ada 95 meta-schema called *Ada_For_Cheddar_Meta_Model*. It contains five entities, *class_in_package*, *attribute*, *string_type*, *real_type* and *in_package_type_alias* and one type, *attr_domain*:

- *class_in_package* specifies a *Cheddar* tagged record; it has four explicit attributes, *super* for the supertype reference, *name* for the name of the tagged record, *attributes*, a list that contains tagged record attribute references and *is_private* that is set to *true* if the tagged record is a private one;
- *attribute* specifies what a tagged record attribute is; it has two explicit attributes, associating a name with a domain;
- *in_package_type_alias* specifies a *Cheddar* subtype with two explicit attributes, *name* and *alias_name*;
- *string_type* and *real_type* specify two basic Ada types;
- and *attr_domain* is defined in order to precisely enumerate attribute domain possible types; an attribute value can be a string, a real, ...

These entities show how to produce the Ada code declaring subtypes and tagged records. Entity *class_in_package* is specified with four translation rules (derived attributes), *with_use_list*, *ptr_type*, *ads_code* and *adb_code*.

As an example, *with_use_list* is intended to produce the list of packages name on which a tagged record is dependent and *ads_code* is intended to produce class definition code in a package by *class_in_package_ads_code* function computing.

3.2 Ada code generation

As shown in Figure 4, given a *Cheddar* model (eg. *Cheddar_Task*, Figure 7), code generation is made of two processes:

```
SCHEMA Ada_For_Cheddar_Meta_Model;

 ENTITY class_in_package;
  super : OPTIONAL class_in_package;
  name : STRING;
  attributes : LIST of attribute;
  is_private : BOOLEAN;
 DERIVE
  with_use_list : LIST OF STRING :=
   class_in_package_with_use_list(SELF);
  ptr_type : STRING := name + '_Ptr';
  ads_code : STRING :=
   class_in_package_ads_code(SELF);
  adb_code : STRING :=
   class_in_package_adb_code(SELF);
 END_ENTITY;

 ENTITY attribute;
  name : STRING;
  domain : attr_domain;
 END_ENTITY;

 TYPE attr_domain =
  SELECT (string_type, real_type, ...);
 END_TYPE;

 ENTITY string_type ...
 ENTITY real_type ...
 ...
 ENTITY in_package_type_alias;
  name : STRING;
  alias_name : STRING;
 DERIVE
  ads_code : STRING :=
    'subtype ' + name
    +' is ' + alias_name + ';';
 END_ENTITY;

 FUNCTION class_in_package_ads_code
     (cip: class_in_package): STRING;
 LOCAL
   code : STRING;
 END_LOCAL;
 code := 'type ' + cip.name + ' is new '
         + cip.super.name + ' with ';
 IF ( cip.is_private ) THEN
  code := code + 'private;\n';
 ELSE
  code := code + attributes_ads_code(cip);
 END_IF;

 ...
 RETURN(code);
 END_FUNCTION;

END_SCHEMA;
```

Figure 3: A part of an Ada/Cheddar meta-model written with EXPRESS

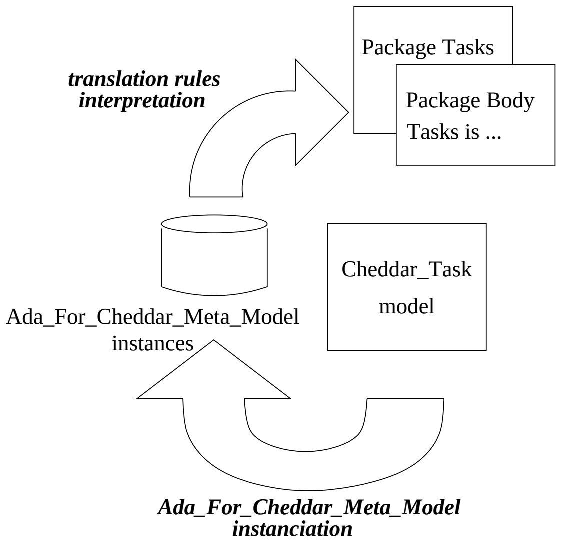

translation rules interpretation
Package Tasks
Package Body Tasks is ...
Cheddar_Task model
Ada_For_Cheddar_Meta_Model instances
Ada_For_Cheddar_Meta_Model instanciation

Figure 4: Code generation process

1. the model is parsed, and as a result, *Ada_For_Cheddar_Meta_Model* meta-model instances are created;

2. translation rules specified in *Ada_For_Cheddar_Meta_Model* meta-model can be interpreted, as a result, an Ada package is generated.

3.3 Reusing Platypus

Even within a meta-environment, meta-model specification, modeling dialogs elaboration or model analyzer implementation are difficult and expensive tasks. The core part of *Platypus* is made of an *EXPRESS* meta-model and of an *EXPRESS* modeling environment. Within *Platypus*, *EXPRESS* models are stored into a repository of models in which models are instances of *Platypus EXPRESS* meta-model.

New domain meta-model (eg. *Ada_For_Cheddar_Meta_Model*) can be defined as *Platypus EXPRESS* meta-model specialization. In such a case, a dialect of *EXPRESS* can be used as the modeling language. *Platypus* modeling environment can be used for meta-modeling as well as for modeling. Then, *Platypus* model to meta-model instanciation procedures can be reused. Using this feature avoid domain specific modeling environment implementation and fully automates application generator running.

3.3.1 Platypus meta-model reuse

EXPRESS is an hybrid object oriented modeling langage. *Platypus EXPRESS* meta-model specify langage and environment concepts that can be reused for another langage modeling. As an example, Figure 5 shows a part of the *Platypus* meta-model, *platypus_dictionary_schema*. It contains the *entity_definition* meta-entity that specify what an *EXPRESS* entity is: mainly, a named type, owned by a context, that may have some supertypes and that may be associated with some attributes.

Figure 6 shows how the *Ada_For_Cheddar_Meta_Model* meta-model is simplified if it is defined as a *Platypus* meta-model specialization:

```
SCHEMA platypus_dictionary_schema;

 ENTITY named_type
  ABSTRACT SUPERTYPE
  SUBTYPE OF ( dictionary_instance );
  name : STRING;
  where_rules : LIST OF where_rule;
  owner : context_definition;
 END_ENTITY;

 ENTITY entity_definition
  SUBTYPE OF ( named_type );
  supertype_constraint :
    OPTIONAL supertype_constraint;
  supertypes : LIST OF
    UNIQUE entity_definition_reference;
  attributes : LIST OF
    UNIQUE attribute;
  uniqueness_rules : LIST OF
    UNIQUE uniqueness_rule;
  complex : BOOLEAN;
  instantiable : BOOLEAN;
  independent : BOOLEAN;
 END_ENTITY;
 ...
```

Figure 5: A part of Platypus EXPRESS meta-model showing entity_definition meta-entity

- *class_in_package* is defined as a subtype of *entity_definition*; now it only has the explicit attribute *is_private*, because privacy does not exist for an entity within *EXPRESS*;

- *in_package_type_alias* is also defined as a subtype of *entity_definition* and keeps the only one explicit attribute (called *alias_name*);

- *attribute*, *attr_domain*, *string_type* and *real_type* are not needed anymore because all these concepts are fully reused from *Platypus* meta-model.

3.3.2 *Platypus environment reuse*

As we explained in section 3.2, application generator is made of two parts.

For a particular meta-model, using *STEP* implementation methods, *Platypus* automates the second part with meta-data checking and translation rules interpretation.

The second part is also automated because *Ada_For_Cheddar_Meta_Model* is specified as a specialization of *Platypus EXPRESS* meta-model. Then our dialect of *EXPRESS* is used as the modeling language and *Platypus* modeling environment is used for modeling *Cheddar* concepts conforming to *Ada_For_Cheddar_Meta_Model* meta-model.

Figure 7 shows the *Cheddar* model called *Cheddar_Task*. *Generic_Task EXPRESS* entity describes how such *Cheddar* feature has to be implemented in Ada (see Figure 11). As an example, *Generic_Task* entity is explicitly linked to its meta-entity *class_in_package*. As a result, in the context of an *EXPRESS* to Ada for *Cheddar* translation, a *Generic_Task*

```
SCHEMA Ada_For_Cheddar_Meta_Model;
 USE FROM platypus_dictionary_schema;

 ENTITY class_in_package
  SUBTYPE OF ( entity_definition );
  is_private : BOOLEAN;
 DERIVE
  super : class_in_package :=
   supertypes[1].ref;
  with_use_list : LIST OF STRING :=
   class_in_package_with_use_list(SELF);
  ptr_type : STRING := name + '_Ptr';
  ads_code : STRING :=
   class_in_package_ads_code(SELF);
  adb_code : STRING :=
   class_in_package_adb_code(SELF);
 WHERE
  have_one_supertype : SIZEOF(supertypes) = 1;
 END_ENTITY;

 ENTITY in_package_type_alias
  SUBTYPE OF ( entity_definition );
  alias_name : STRING;
 DERIVE
  ads_code : STRING :=
   'subtype ' + name
   +' is ' + alias_name + ';';
  adb_code: STRING:= '';
 END_ENTITY;
  ...
```

Figure 6: A part of an Ada/Cheddar meta-model reusing the Platypus meta-entity entity_definition

is not only an entity definition but also a tagged record in an Ada package.

4. SOURCE CODE AUTOMATICALLY PRODUCED WITH THE ADA META-MODEL

4.1 Few words about the Cheddar design

Let see now which *Cheddar* source code we plan to generate from our meta-model. *Cheddar* is a set of Ada packages which aims at performing performance analysis of concurrent real time applications. *Cheddar* is composed of two software components:

- a framework which implements the analysis methods and algorithms,

- a *GtkAda* machine-man interface which provides an easy way to call the framework sub-programs and to display the analysis results.

These two components are written with a container library which can be configurated in order to use static or dynamic memory allocations.

With *Cheddar*, a real time application is modeled as sets of processors, address spaces, buffers, resources, and tasks. Attributes of such features are stored in simulation data.

```
SCHEMA Cheddar_Task;
 META FROM Ada_For_Cheddar_Meta_Model;

 TYPE Policies {ada_enumeration (?) }
  = ENUMERATION
     OF (Sched_Fifo, Sched_Rr, Sched_Others);
 END_TYPE;

 ENTITY Generic_Task{class_in_package(false)}
   SUBTYPE OF ( Generic_Object );
      Task_Policy : Policies;
        ...
 END_ENTITY;

 ENTITY Periodic_Task{class_in_package(false)}
   SUBTYPE OF ( Generic_Task );
     Period : Natural_Type;
        ...
 END_ENTITY;

END_SCHEMA;
```

Figure 7: A part of the Ada/Cheddar domain model specified with Platypus EXPRESS dialect

The framework provides services to manage *Cheddar* simulation data. For each simulation data type, the framework implements sub-programs to perform integrity checks, to print/parse XML or AADL[21] specification files and to store them in containers. Finally, the machine-man interface provides a widget for each feature in order to update containers and to get feature attributes.

For each *Cheddar* feature, we expect to generate three packages (see Figures 10, 11 and 12):

1. each *Cheddar* feature is implemented by an Ada class. For example, the Ada class corresponding to the task feature is composed of the tagged records *Generic_Task* (the super tagged record of the task class) and its derived tagged records (eg. *Periodic_Task*, *Aperiodic_Task*, ...). This first generated package includes such a feature declarations (see Figure 11);

2. the set of instances of an Ada class (eg. instances of *Generic_Task*, *Periodic_Task*, *Aperiodic_Task*, ...) is stored in a container. This container is built from a generic package which is extended to provide input/ouput sub-programs. For example, this package implements AADL/XML printer, parser and integrity checks sub-programs. *Cheddar* can be currently compiled with two different Ada 95 container implementations: the first implementation only does static memory allocations and the second one is based on dynamic memory allocations. The user has the possibility to choose one of these implementations. This choice has be taken into account by the meta-model. Figure 12 shows an example of such a container package;

3. a package containing a *GtkAda* Widget will be also generated (see Figure 10). The window of this widget is split in two sub-windows: the top-left sub-window allows the user to get feature attributes. The bottom-left sub-window displays a set of buttons which can be pressed to update the container. The instances stored in the container are listed in the right sub-window.

Software components	Size	Expected to be generated
Containers and configuration packages	7205	few lines (configuration)
Machine-Man interface	53650	17013 (31,7%)
Simulation framework	79297	27474 (34,6%)
All components	140152	44487 (25-35%)

Figure 8: Number of lines of the Cheddar toolset

4.2 Current status and first results

The design of the Ada meta-model and the model of the *Cheddar* library is still in progress. At the time we write this article, only the Ada packages implementing *Cheddar* features are automatically generetad.

The *EXPRESS* source code modeling the *Cheddar* features and their data is composed of 348 lines of code. 947 lines of *EXPRESS* source code were required in order to write the Ada 95 meta-model (this amount of lines also includes the Ada source code generator). This meta-model can be reuse for the modeling of any Ada 95 object oriented application.

The amount of automatically generated source code for this part of *Cheddar* is about 75 percents of the Ada packages which were originally manually implemented. But, when the refactoring work will be over, in the best case, about 30 percents of the *Cheddar* library can be expected to be automatically implemented. The Figure 8 gives an overview of the Ada code we expect to automatically produce with the meta-model.

The current Ada meta-model describes the features of record (discriminated or not), tagged record (with or without private types and with or without tagged record extension), enumeration, constrained array types and generic packages instanciation. For each of these Ada concepts, the Ada source code generator produces the type definition but also an access type definition and a sub-program to release dynamically allocated memory. For each record and tagged record types, the Ada source code generator also produces sub-programs to perform basic input/output operations on the type (eg. *Put* sub-programs), to initialize and finalize objects (if the tagged record extends *Ada.Finalization.Controlled* tagged record) and to provide specific *Cheddar* services. From the meta-model, *renames* and *subtypes* can also be generated if necessary. Finally, by expressing *Cheddar* feature relationships, the *use* and *with* clauses are also automatically computed. Of course, the Ada 95 meta-model and the Ada source code generator can be adapted to the designer requirements.

Using CASE tools (with or without meta-modeling capabilities) also improves the design of the target application. By defining the Ada 95 meta-model and the *Cheddar* model, we were able to detect the following mistakes in the previous *Cheddar* design and implementation :

- some anti-patterns were detected and removed (eg. a violation of the open-close principle in the *Cheddar* features copy constructor[15]);

- some sub-programs were defined in wrong packages (packages which do not contain the types related to the misplaced sub-programs). For instance, the sub-programs which check *Cheddar* feature integrity constraints were defined in the machine-man interface part. Then, no integrity check was able to be performed when *Cheddar* data were provided from AADL files;
- the way identifiers were built was sometimes wrong (identifiers of *GtkAda* sub-widgets, identifiers of tagged record, of access types, of enumeration sub-programs, ...). This mistake was easily corrected with the Ada source code generator;
- some un-used attributes in the *GtkAda* user-defined widget were detected. Some un-usefull *use/with* clauses was also detected and removed (extra *use/with* clauses in child packages);
- some basic sub-programs, subtypes and renames on *Cheddar* features were missing. The automatic generation of such code eases the use of the *Cheddar* library.

Finally, using meta-modeling will decrease the future maintenance cost. Basically, adding a new analysis tool into *Cheddar* requires to add new attributes in the existing *Cheddar* features. Let take the example of a new kind of scheduling algorithm that we added last year: this scheduling algorithm, called the MUF scheduler[25], assumes that a criticality level is defined for each task of the system. Adding this new attribute into the tagged record which models a task in the *Cheddar* framework leads to the modification of 8 Ada packages. 90 lines of the Ada code were written to store, initialize, and display this new attribute in the machine-man interface. By the use of the Ada meta-model, most of these 90 lines of code may be automatically implemented.

5. CONCLUSION

In this paper, we have presented the refactoring work of *Cheddar*, a set of Ada packages which aims at providing performance analysis tools for concurrent real time applications. CASE tools can be used for such a purpose. However, we chose to use a meta CASE tool called *Platypus*. Indeed, it seems that few studies exist concerning Ada and meta-modelization. Here, we're investigating how to use a meta CASE tool in order to automatically produce some parts of an Ada 95 object oriented software.

At the time we write this article, the Ada 95 meta-model expresses the most important Ada 95 object-oriented features that we use in *Cheddar*. It is written in *EXPRESS*. From this meta-model, we were able to describe the *Cheddar* data model and we developed the Ada code generator to automatically produce a part of the *Cheddar* implementation. The amount of automatically generated source code for this part of *Cheddar* is about 75 percent of the Ada packages which were originally manually implemented. But, when the refactoring work will be over, in the best case, around 30 percents of the *Cheddar* library can be expected to be automatically implemented. This refactoring work also increased the quality of the *Cheddar* design.

This level of automatically generated source code can be achieve with classical CASE tools without meta-modeling capabilities. The next step will consist in including in the meta-model some concepts which are more specific to the *Cheddar* domain such as the scheduling algorithms currently implemented in the analysis tools of *Cheddar*. Such an improved Ada 95 meta-model should make it possible to generate a part of the scheduling simulation engine of *Cheddar*. This part of code could not be designed and generated with a CASE tool without meta-modeling capabilities.

6. REFERENCES

[1] A. Burns and A.J. Wellings. HRT-HOOD: A Design Method for Hard Real-time Systems. *Real Time Systems journal*, 6(1):73–114, 1994.

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

[3] Dome Official Website. http://www.htc.honeywell.com/dome/download.htm.

[4] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. TOPCASED : An Open Source Development Environment for Embedded Systems. *Chapter 11, From MDD Concepts to Experiments and Illustrations, ISTE Editor*, pages 195–207, September 2006.

[5] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-time Uni-processor Scheduling. INRIA Technical report number 2966, 1996.

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

[7] SAE Inc. 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 0.994, August 2004.

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

[9] ISO 10303-11. *Part 11: EXPRESS Language Reference Manual*, 1994.

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

[11] 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.

[12] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real Time Analysis*. Kluwer Academic Publishers, 1994.

[13] Yi Lu. Reading project report, comparison of meta-modelling techniques and tools. Technical report, Computer Science Department, McGill University, March 2003.

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

[15] B. Meyer. Object Oriented Software Constructions. *Prentice Hall editor*, 2000.

[16] Mourad El-Hadj Mimoune, Guy Pierra, and Yamine Ait-Ameur. An ontology-based aproach for exchanging data between heterogeneous database systems. In

ICEIS 2003: Proceedings of the 5th International Conference On Enterprise Information Systems, Angers - France, 2003. École Supérieure d' Électronique de l'Ouest.

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

[18] A. Plantec and V. Ribaud. Experiences using an Application Generator Builder. *Proceedings of the 11th International Conference on software engineering and knowledge engineering, June the 16-19, Kaiserslautern, Germany*, 1999.

[19] Platypus Technical Summary and download. http://cassoulet.univ-brest.fr:8000/Platypus.

[20] J. Regehr and J. A. Stankovic. Hls : a framework for composing soft real-time schedulers. *In the 22th IEEE International Real-Time Systems Symposium (RTSS'01). London, UK.*, pages 3–14, December 2001.

[21] SEI. OSATE : An extensible Source AADL Tool Environment. *SEI AADL Team technical Report*, December 2004.

[22] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a Flexible Real Time Scheduling Framework. International ACM SIGADA Conference, Atlanta, USA, November 2004.

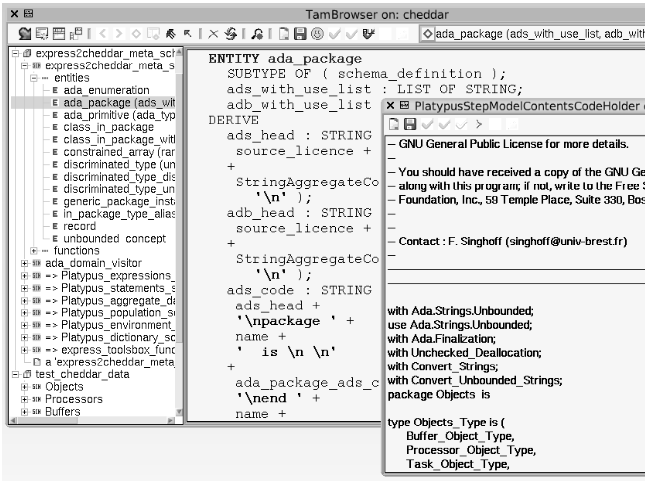
[23] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and Memory requirements analysis with AADL. International ACM SIGADA Conference, Atlanta, USA, November 2005.

[24] Squeak web site. http://www.squeak.org.

[25] D. B. Stewart and P. K. Khosta. Real-Time Scheduling of Dynamically Reconfigurable Systems. *In Proceedings of the IEEE International Conference on Systems Engineering, Dayton, Ohio*, pages 139–142, August 1991.

[26] EMF website. Eclipse Modeling Framework. *http://www.eclipse.org/emf*.

7. ANNEX



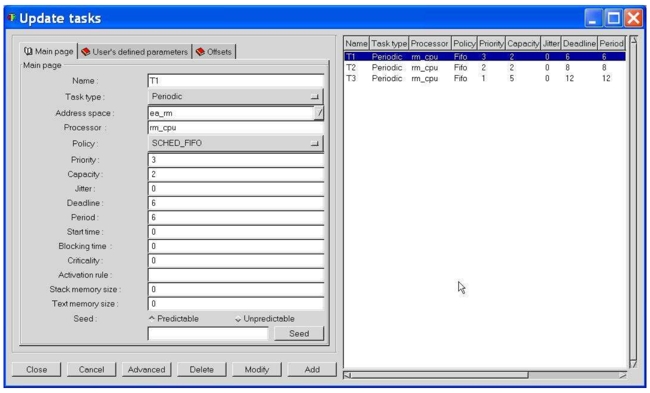
```
ENTITY ada_package
  SUBTYPE OF ( schema_definition );
  ads_with_use_list : LIST OF STRING;
  adb_with_use_list
DERIVE
  ads_head : STRING
   source_licence +
  +
   StringAggregateCo
     '\n' );
  adb_head : STRING
   source_licence +
  +
   StringAggregateCo
     '\n' );
  ads_code : STRING
   ads_head +
   '\npackage ' +
   name +
   '  is \n \n'
  +
   ada_package_ads_c
   '\nend ' +
   name +
```

```
– GNU General Public License for more details.
–
– You should have received a copy of the GNU Ge
– along with this program; if not, write to the Free
– Foundation, Inc., 59 Temple Place, Suite 330, Bos
–
–
– Contact : F. Singhoff (singhoff@univ-brest.fr)
–


with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;
with Ada.Finalization;
with Unchecked_Deallocation;
with Convert_Strings;
with Convert_Unbounded_Strings;
package Objects  is

type Objects_Type is (
    Buffer_Object_Type,
    Processor_Object_Type,
    Task_Object_Type,
```

Figure 9: A Platypus snapshot: the deepest window shows the Platypus editor with, on the left, the tree of EXPRESS elements corresponding to the current schema edited on the right. The front window shows a code generating result.



Main page

Name :	T1
Task type :	Periodic
Address space :	ea_rm
Processor :	rm_cpu
Policy :	SCHED_FIFO
Priority :	3
Capacity :	2
Jitter :	0
Deadline :	6
Period :	6
Start time :	0
Blocking time :	0
Criticality :	0
Activation rule :	
Stack memory size :	0
Text memory size :	0
Seed :	Predictable / Unpredictable

Name	Task type	Processor	Policy	Priority	Capacity	Jitter	Deadline	Period
T1	Periodic	rm_cpu	Fifo	3	2	0	6	6
T2	Periodic	rm_cpu	Fifo	2	2	0	8	8
T3	Periodic	rm_cpu	Fifo	1	5	0	12	12

Figure 10: Widget related to a Cheddar feature.

```
with Text_Io;
use Text_Io;
with Unchecked_Deallocation;
with Convert_Strings;
with Convert_Unbounded_Strings;
with Objects;
use Objects;

package Tasks is

  type Policies is (Sched_Fifo,...
  procedure To_Policies is
    new Convert_Strings(Policies, Sched_Fifo);
  procedure To_Policies is
    new Convert_Unbounded_Strings(Policies,
    Sched_Fifo);
  package Policies_Io is new
    Text_Io.Enumeration_Io(Policies);
  use Policies_Io;

  type Generic_Task is
    abstract new Generic_Object with
  record
    Policy            : Policies;
    ...
  end record;

  type Generic_Task_Ptr is
    access all Generic_Task'Class;
    ...

  type Periodic_Task is
    new Generic_Task with
  record
      Period : Natural;
      ...
  end record;
  type Periodic_Task_Ptr is
    access all Periodic_Task'Class;

  procedure Initialize
      (A_Task : in out Periodic_Task);
  function Copy
      (A_Task : in Periodic_Task_Ptr)
    return Periodic_Task_Ptr;
  function Copy
      (A_Task : in Periodic_Task)
    return Periodic_Task_Ptr;
  procedure Put
    (A_Task : in Periodic_Task_Ptr);
  procedure Put
    (A_Task : in Periodic_Task);
  procedure Free is
    new Unchecked_Deallocation
      (Periodic_Task'Class,
       Periodic_Task_Ptr);
  ...
end Tasks;
```

Figure 11: Part of a package specification generated for the Cheddar feature. *Policies* is an attribute type generated from an *EXPRESS* type. *Generic_Task* and *Periodic_Task* are tagged records generated from *EXPRESS* entities.

```
with Sets;
with Tasks;
use Tasks;

package Task_Set is

  package Generic_Task_Set is
    new Sets (Element => Generic_Task_Ptr,...);
  type Tasks_Set is
    new Generic_Task_Set.Set with private;
  subtype Tasks_Range is
    Generic_Task_Set.Element_Range;
  ...

  -- XML/AADL printer sub-programs
  function Export_Xml(My_Tasks : Tasks_Set
      ...) return Unbounded_String;
  function Export_Aadl(My_Tasks : Tasks_Set
      ...) return Unbounded_String;

  -- Perform integrity checks on attributes
  procedure Check_Integrity
      (My_Tasks : in Tasks_Set...);
  ...
end Task_Set;
```

Figure 12: Example of a container package which stores instances of a feature. *Tasks_Range* is a subtype generated from an *EXPRESS* entity.