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 toolkit 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 Computer-Aided Software Engineering 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.

Copyright ACM (2006). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the ACM SIGADA 2006 International conference Proceedings, November 12–16, 2006, Albuquerque, New Mexico, USA.
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-model. 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 database 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 made 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
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 predefined 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 instantiation 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 development 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 Speake[34], a free Smarttalk 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[5] 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 multi-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 (e.g. Ada source code, documentation, XML data).

Platypus meta-model can itself be re-used 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;
- 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 allow how to produce the Ada code declaring subtypes and tagged records. Entity class_in_package is specified with four translation rules (derived attributes), with_in_list, plast_type, ada_code and ada_code. As an example, with_in_list is intended to produce the list of packages name on which a tagged record is dependent and ada_code is intended to produce class definition code in a package by class_in_package, ada_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:
3.3 Reusing Platypus

Even within a meta-environment, meta-model specification, modeling dialog 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 metamodel.

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 instantiation 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 language. Platypus EXPRESS meta-model specify langage and environment concepts that can be reused for another language 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:
**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 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.

**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 *Ada* 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 configured 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.
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/output 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 been 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

Figure 8: Number of lines of the Cheddar toolset

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.

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 generated.

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 reused 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 percent of the Ada packages which were originally manually implemented. But, when the refactoring work will be over, in the best case, about 30 percent 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 instantiation. 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, names 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 codes uses 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[23], 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 model-based design. 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 percentages 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

7. ANNEX

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.

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;

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 sub-type generated from an EXPRESS entity.

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.