Towards User-Level extensibility of an Ada library : an experiment with Cheddar

Frank Singhoff, Alain Plantec

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

Abstract. In this article, we experiment a way to extend an Ada library called Cheddar. Cheddar provides a domain specific language. Programs written with this domain specific language can be interpreted in order to perform real time scheduling analysis of real time systems. By the past, different projects showed that the Cheddar programming language is useful for the modeling of real time schedulers. But these experiments also showed that the interpreter is lacking of efficiency in case of large scheduling simulations. In this article, by designing a Cheddar meta-model, we investigate on how to compile such Cheddar programs in order to extend the Cheddar library. For such a purpose, we use *Platypus*, a meta CASE Tool based on *EXPRESS*. For a given Cheddar program and with a meta-model of Cheddar handled by *Platypus*, we can generate a set of Ada packages. Such Ada packages can be compiled and integrated as builtin schedulers into Cheddar. Then, the efficiency of scheduling simulations can be increased.

Key words: Meta-modeling, Ada code generating, Cheddar, Platypus

1 Introduction

This article deals with the Cheddar library [1]. Cheddar is a library designed for the performance analysis of real time applications. With Cheddar, a real time application is modeled as a set of tasks, processors, schedulers, buffers ... This library provides a set of real time schedulers and their analysis tools implemented in Ada. Schedulers currently implemented into Cheddar are mostly met in real time applications and the library can be used to perform performance analysis of many different types of real time applications. However, it exists a need to extend these Cheddar analysis tools to specific scheduler or task models. For such a purpose, it requires that the user understands the Cheddar design. Furthermore, designing a new scheduler or a new task model may be difficult without an environment which makes it possible to easily write and test the scheduler code. In order to ease the design of new schedulers, Cheddar provides a programming language. The model of a scheduler or of a task model described

and tested with the Cheddar programming environment is interpreted : thus the designer can easily handle and experiment his scheduler models.

Different projects showed that the Cheddar programming language is useful for the modeling of real time schedulers. But these experiments also showed that the interpreter is lacking of efficiency in case of large scheduling simulations.

This article presents a way to extend an Ada library. It presents a way to compile Cheddar programs by automatically producing Ada code corresponding to the modeled schedulers. For such a purpose, we use the *Platypus* meta CASE tool and a meta-model which specifies both the Ada 95 programming features and the Cheddar programming language features. Then, from the user-defined scheduler model, one can generate the new builtin scheduler fully integrated into the Cheddar library.

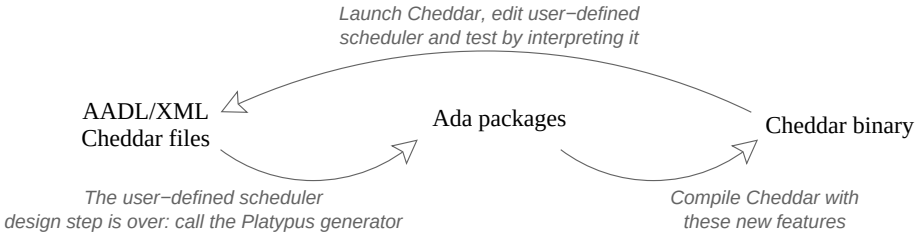

Fig. 1. The user-defined scheduler design process with Cheddar

The scheduler design and integration process proposed in this article is depicted by figure 1. This process is made of two main steps:

1. the first step is the new scheduler or the new task model design; a new scheduler and task model is specified with the Cheddar programming language; such a program can be interpreted by Cheddar allowing early testing and improvements;
2. the second step is the new scheduler or the new task model integration. It consists in the automatic generation of Ada packages from the user-defined scheduler; a Cheddar version integrating the new scheduler and task model is then recompiled.

This article is organized as follows. In section 2, we outline the Cheddar programming language and we give an example of its use. Then, we shortly describe the design of the Cheddar library in section 3. Section 4 is devoted to the process we use to generate Ada packages with the meta CASE Tool *Platypus*. In section 5, we propose a meta-model for the Cheddar programming language. The generated Ada code is briefly described in section 6. Finally, we conclude by describing the current status of the project in section 7.

2 The Cheddar programming language

In this section, we give an overview of the Cheddar programming language. A complete description of the language can be found in [2]. The use of the Cheddar programming language is illustrated by the example of the sporadic task model with an EDF scheduler [3].

2.1 Outline of the Cheddar programming language

The Cheddar programming language is composed of two parts :

1. a small Ada-like language which is used to express computations on simulation data. Simulation data are constants or variables used by the scheduling simulator engine of Cheddar in order to simulate task and scheduler behaviors. Examples of simulation data are task wake up times or task priorities. A program written with the Cheddar programming language is organized in sections. A section is a kind of Ada sub-program;
2. a set of timed automata such as those proposed by UPPAAL [4–6] which is a toolbox for the verification of real time systems. In Cheddar, these timed automata allow to model timing and synchronization behaviors of tasks and schedulers. Automata may run Ada-like sections in order to read or modify simulator data.

The Ada-like language provides usual statements such as loops, conditionnal statements, assignments. It also provides statements which are specific to the design and the debug of scheduling algorithms. Two kinds of such specific statements exist: high-level and low-level statements. High-level statements operate on vectors which store a simulation data for all tasks, messages, buffers, processors ... Low-level statements only operate on scalar simulation data.

The simplified BNF syntax of the language is given in figure 8. The *entry* rule specifies that a program is a set of sections. Most of the time, a program is composed of the following sections [1]: a *start_section* which contains variable declarations ; a *priority_section* which contains the code to compute simulation data on each unit of time during simulation; and an *election_section* which looks for the task to run on simulation time.

The *statement* rule gives the syntax of all available statements to the scheduler designer. The most important specific statements are the *return* statement and the $uniform/exponential$ statements. The *return* statement gives the identifier of the task to run. $uniform/exponential$ statements customize the way random values are generated during simulation time.

The language also provides operators and types. It provides usual Ada types such as scalar *integer*, *boolean*, *double*, *string* types or array and their attributes ($first$, *last*, *range*, ...). It also provides usual logical and arithmetic operators. As for the statements, scheduling specific types and operators are available. For instance, the *lcm* operator computes the last common multiplier of simulation data, the *max_to_index* operator looks for the ready task which has the highest

value of an array variable and the *random* type provides random generator capabilities.

The second part of a scheduler or a task model is a specification of its timing and synchronization behavior with a set of automata. Automata used by Cheddar are timed automata as they are defined in the UPPAAL toolset [6]. A timed automaton is a finite state machine extended with clock variables. In UPPAAL, a system is modeled as a network of several timed automata. The model is extended with variables. A state of the system is defined by the locations of all automata, the clock constraints, and the values of the variables. Every automaton may fire a transition separately or synchronize with another automaton, which leads to a new state.

At least, each automaton has to be composed of some predefined locations and transitions. Transitions can run sections, read and write simulation data depending on the statements given by users on transitions (clock update, synchronization, guard, ...). The BNF syntax of a guard, a synchronization or a clock update can be read in [6].

2.2 Example of a user-defined task model : the sporadic task model

```
: start_section
   dynamic_priority : array (tasks_range) of integer;
   gen1 : random;
   exponential(gen1, 100);
   cycle_duration : array (tasks_range) of integer;

sporadic_model : task_activation_section
   cycle_duration:=max(tasks.period, gen1);

: priority_section
   dynamic_priority := tasks.start_time
     + ((tasks.activation_number-1)*tasks.period)
     + tasks.deadline;

: election_section
   return min_to_index(dynamic_priority);

sporadic_model : automaton_section
  Initialize : initial state;
  Pended, Ready, Blocked, Run : state:

  Initialize --> [ ,tasks.activation_number:=0 ,start_section?] --> Pended
  Pended --> [ engine_clock>=tasks.activation_number+
    cycle_duration, cyclic_task_clock:=tasks.capacity, ] --> Ready
  Ready --> [ , cyclic_task_clock:=cyclic_task_clock+1,elect?] --> Run
  Blocked --> [ , , V!] --> Ready
  Run --> [ cyclic_task_clock>0, , preempt!] --> Ready
  Run --> [ ,cyclic_task_clock:=0, task_activation_section!] --> Pended
```

Fig. 2. Example of a user-defined task model : the sporadic task

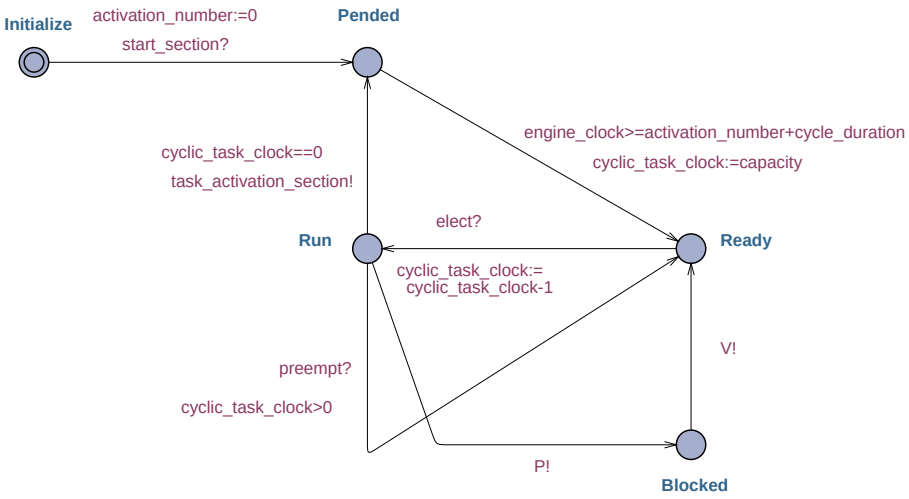

Fig. 3. Cyclic task modeling

Figure 2 and figure 3 show an example of a user-defined task model : the sporadic task model. A sporadic task is a task which can be activated several times and which has a minimal delay between each of its successive activations [7].

Figure 2 shows a Cheddar program modeling a sporadic task. The *task_activation_section* specifies how inter-activation delays have to be computed. In this case, the delay is the maximal value between the period of the task [1] and a value which is randomly generated according to an exponential density function.

The EDF scheduler run tasks according to task deadlines. These deadlines are computed in the *priority_section* and the task with the shortest deadline is chosen in the *election_section* sub-program.

Figure 3 models the synchronization and the timing part of the sporadic task model. In this example, the automaton describes the different task states : basically, a task can be *Pended* (it waits its next activation), *Blocked* (it waits for a shared resource access), *Ready* (it only waits for the processor), or *Run* (currently accessing the processor). The *task_activation_section* is called when the task goes from the *Run* location to the *Pended* location. During this call, the delay that the task has to wait upto its next activation time is computed and stored in the *cycle_duration* variable, as seen in figure 2.

3 Implementation of Cheddar

Before proposing a meta-model of Cheddar, let see how the library is implemented. The library implements the components showed in figure 4:

[1] The period is the minimal inter-activation delay.

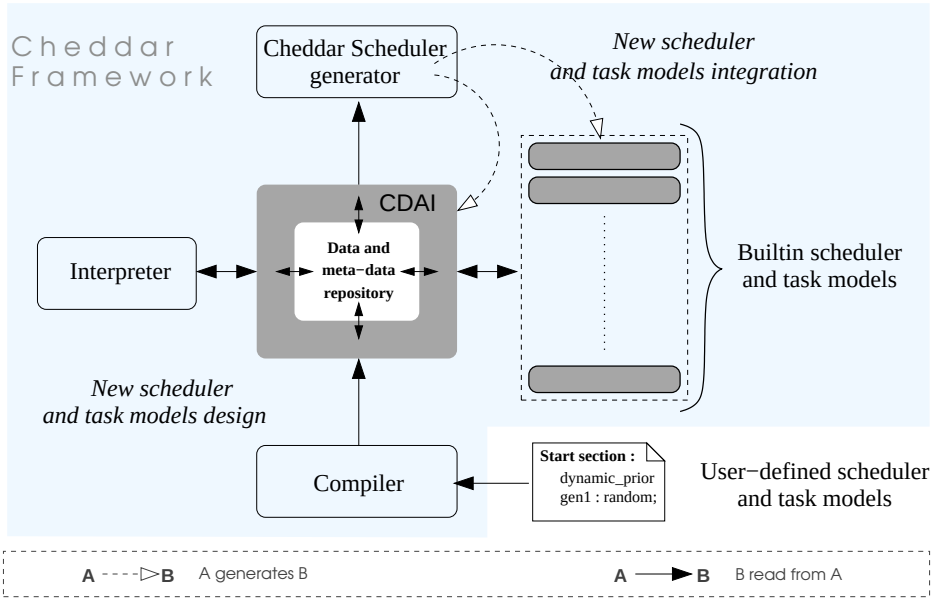

Fig. 4. The Cheddar library

- Cheddar implements a data and a meta-data repository (eg. tasks, processors or schedulers are data; an automaton transition definition or a loop statement is a meta data);
- data and meta-data are read or written from or into the repository using the *Cheddar Data Acces Interface* (CDAI); the CDAI is used by every components of Cheddar library in order to read or write data and meta-data;
- a new scheduler and task model is specified using the Cheddar programming language; the result is a Cheddar program; this program is read by a compiler that produces meta-data constituting an internal representation for it; meta-data are stored into the repository;
- the Cheddar interpreter is implemented in order to run user-defined schedulers; an interpreting process run statements and expressions stored as meta-data and interacts with Cheddar for data values reading and writing;
- the Cheddar scheduler generator is used to produce Ada packages from user-defined schedulers; for one user-defined scheduler, generated code consists in :
  - a package that implements the scheduler;
  - a package that extends CDAI for the new scheduler.

4 Modeling, meta-modeling and code generating

Data and meta-data stored into the repository are described by a set of models and meta-models. All these models are handled by specific Ada code generators

built with the meta CASE tool *Platypus* [8]. First, this section briefly describes the *Platypus* meta-CASE tool. Second, it describes *Platypus* using for code generating.

4.1 The Platypus meta CASE tool

Platypus [8] is a meta-environment fully integrated inside *Squeak* [9], 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 *ISO 10303* namely the *STEP* [10] standard for meta-models specification and implementation. *STEP* defines a dedicated technology, mainly an object oriented modeling language called *EXPRESS* [11] that can be used as a modeling language as well as a meta-modeling language [12, 13].

In *Platypus*, a meta-model consists in a set of *EXPRESS* schemas that can be used to describe a language. The main components of a meta-model are types and entities. They are describing the language features. Entities contain a list of attributes that provide buckets to store meta-data while local constraints are used to ensure meta-data soundness.

Code generators are specified by translation rules. 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).

4.2 Code generating

As shown by figure 5, code generation is used at two levels of abstraction:

1. the first level is the Cheddar level. This level is related to a particular Cheddar version for which all handled object types (processors, tasks, buffers, ...) are fixed and described by the Cheddar model. The Cheddar model is an *EXPRESS* model, it is parsed by an Ada code generator that produces the CDAI set of packages. Translation rules applied by the generator are specified by an Ada for Cheddar meta-model. More explanations about this first level of design can be read in [14];
2. the second level is the Cheddar language level. It corresponds to Cheddar specializations driven by the specification of new schedulers and task models. These new parts are specified using the Cheddar programming language. The dedicated code generator is able to parse a Cheddar program and first, to produce a new scheduler implementation and second, to enrich the Cheddar object model. Then, the CDAI packages are regenerated and a new Cheddar version can be compiled.

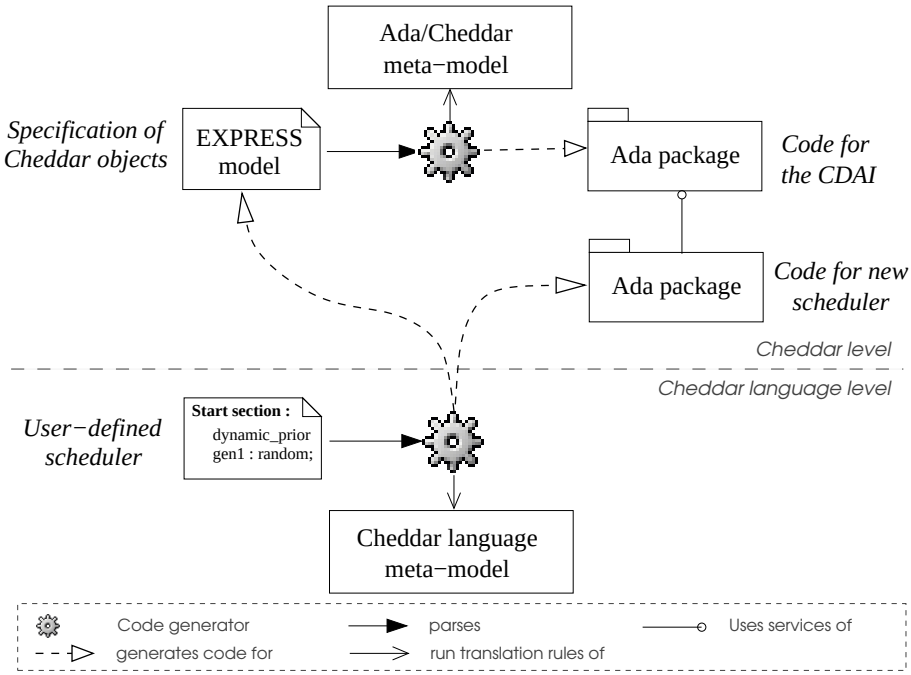

Fig. 5. Cheddar code generating from models and meta-models

5 Meta-modeling of the Cheddar programming model

The Cheddar programming language model is made of one meta model specified with *EXPRESS* that includes meta-data types (meta-entities), their relationships, constraints, and includes translation rules that are specified as meta-entities computed properties.

The meta-model is made of several *EXPRESS* schemas. *cheddar_language* and *cheddar_automaton* schemas are the two main schemas. A part of their specification is shown in figure 6. *cheddar_language* includes the *cheddar_schema* entity that specifies what a Cheddar program is. A program is made of a set of sections. Two main kind of section can be specified, a *program_section* or an *automaton_section*. A *program_section* mainly contains a list of statements. An *automaton_section* is made of the specification of an automaton.

Cheddar compiler populates this meta-model. As an example, from one Cheddar program, one instance of *cheddar_schema* entity is created and stored into the repository.

Code generating consists in the reading of the two translation rules specified in *cheddar_schema* entity, namely the two derived attributes *cdai_entities* and *scheduler_package* :

– *cdai_entities* value is computed by the evaluation of the function *cheddar_schema_cdai_entities*; computing result consists in a new *EXPRESS*

```
SCHEMA cheddar_language;
    ENTITY cheddar_schema;
           sections : SET [1:?] OF section_definition;
       DERIVE
           cdai_entities : STRING:=cheddar_schema_cdai_entities(SELF);
           scheduler_package : STRING:=cheddar_schema_scheduler_package(SELF);
       WHERE
           have_one_and_only_one_election :
           sizeof(query(e <* sections | e.identifier = 'election_section'))=1;
    END_ENTITY;
    ENTITY section_definition ABSTRACT SUPERTYPE;
           identifier : STRING;
       INVERSE
           context : cheddar_schema FOR sections;
       UNIQUE
           context, identifier;
    END_ENTITY;
    ENTITY election_section_definition SUBTYPE OF (section_definition);
           task_id : return_stmt;
       DERIVE
           SELF\section_definition.identifier:STRING:='election_section';
    END_ENTITY;
    ENTITY program_section_definition SUBTYPE OF (section_definition);
           declarations : LIST OF variable;
           statements : LIST OF statement;
    END_ENTITY;
    ENTITY automaton_section_definition SUBTYPE OF (section_definition);
           automaton : automaton_definition;
    END_ENTITY;
    ...
SCHEMA cheddar_automaton;
    ENTITY automaton_definition;
          states : SET OF state_definition;
          transitions : SET OF transition_definition;
          initial_state : state_definition; ...
    END_ENTITY;
    ...
```

Fig. 6. A part of the meta-model for the Cheddar programming language

schema which extends the Cheddar model. From this new model we generate a new set of Ada packages which extends the CDAI.

– *scheduler_package* value is computed by the evaluation of the function *cheddar_schema_scheduler_package*; computing result consists in the new scheduler and task model Ada package.

6 Cheddar Ada packages which are generated from a Cheddar program

The Ada packages we generate for the Cheddar level is described in [14]. For the Cheddar language level, two different Ada packages are expected to be generated : packages implementing a new user-defined task model and packages implementing a new user-defined scheduler.

Cheddar tasks are implemented by a set of tagged records (see [14]) : each task type is defined by a tagged record. The *Task_Activation* method of such

```
10  procedure Build_Scheduling_Sequence(...) is
20  ...
30  begin
40   for I in Processor_Range loop
50    Check_Before_Scheduling(...);
60    Scheduler_Initialize(...);
70   end loop;
80
90   while(Current_Time < Total_Scheduling_Time) loop
100     for I in Processor_Range loop
110       Do_Election(...);
120       Next_Task(...);
130     end loop;
140     Current_Time:=Current_Time+1;
150  end loop
160 end Build_Scheduling_Sequence;
```

Fig. 7. Algorithm sketch of the Cheddar simulation engine

a tagged record is able to compute the task wake up times. This sub-program is generated according to the statement the user gives in its Cheddar program (source code provided into the *task_activation_section*).

Cheddar schedulers are also implemented by a set of tagged records. A new scheduler is implemented by extending an abstract tagged record or any already existing schedulers which has a similar behavior. In order to be plugged with the Cheddar simulation engine, each scheduler tagged record has to implement a set of sub-programs such as :

- *Scheduler_Initialize* which initializes variables used by the scheduler; this Ada sub-program contains the *start_section* code;
- some sub-programs to check scheduler assumptions (eg. the *Check_Before_Scheduling* sub-program);
- *Do_Election* which computes task priorities and chose the task to run; such a sub-program contains the *priority_section* and the *election_section* of the implemented scheduler;
- Finally, if a scheduler requires to store data for the tasks it provides scheduling facilities, it has to define a TCB[2] and a set of sub-programs to copy, initialize or display instances of such a TCB.

Figure 7 shows how these sub-programs work all together. The *Build_Scheduling_Sequence* is the main entry point of the Cheddar scheduling simulator. Since a system analyzed by Cheddar may model a multi-processors system, the *Build_Scheduling_Sequence* drives the simulation time unit per time unit. First, simulation data are initialized and some checks are performed to be sure that tasks meet scheduler assumptions (lines 30-70). Then, time unit per time unit, schedulers are called (line 110) and task wake up times are computed (line 120).

[2] TCB stands for Task Control Block.

7 Conclusion

This article describes a way to extend an Ada library. The method is experimented with Cheddar, a library providing performance analysis tools. Cheddar provides a domain specific language which helps users to the design of real time schedulers. Programs written with this domain specific language can be interpreted in order to perform real time scheduling analysis of real time systems. By the past, different projects showed that the Cheddar programming language and its interpreter are useful for the modeling of real time schedulers. But these experiments also showed that the interpreter is lacking of efficiency in case of large scheduling simulations. In this article, we experiment a way to compile such a program. By designing a Cheddar meta-model, we show how to compile Cheddar programs in order to extend the Cheddar library. For such a purpose, we use *Platypus*, a Meta CASE Tool based on *EXPRESS*.

At the time we write this article, the CDAI is modeled and we are able to generate the corresponding Ada packages [14]. We are currently designing the meta-model of the Cheddar programming language and experimenting Cheddar scheduler and task generation.

References

1. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar : a Flexible Real Time Scheduling Framework, ACM Ada Letters journal. 24(4):1-8. Also published in the proceedings of the International ACM SIGAda Conference, Atlanta, USA (2004)
2. Singhoff, F.: Cheddar Release 2.x User's Guide. Technical report, number singhoff-01-2007, Available at http://beru.univ-brest.fr/~singhoff/cheddar (2007)
3. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the Association for Computing Machinery **20**(1) (1973) 46–61
4. Hopcroft, J.E., Ullman, J.D.: Introduction of Automata Theory, Languages and Computation. (2001)
5. Alur, R., Dill, D.L.: Automata for modeling real time systems, Proc. of Int. Colloquium on Algorithms, Languages and Programming, Vol 443 of LNCS (1990) 322–335
6. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. Technical Report Updated the 17th November 2004, (Department of Computer Science, Aalbord University, Denmark)
7. Sprunt, B., Sha, L., Lehoczky, J.: Aperiodic Task Scheduling for Hard-real-time Systems. The Journal of Real Time Systems **1** (1989) 27–60
8. Plantec, A.: Platypus Technical Summary and download. (http://cassoulet.univ-brest.fr/mme)
9. Team, T.S.: Squeak web site. (http://www.squeak.org)
10. ISO 10303-1: Part 1: Overview and fundamental principles. (1994)
11. ISO 10303-11: Part 11: EXPRESS Language Reference Manual. (1994)
12. Plantec, A., Ribaud, V.: 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)

```
entry := sections
sections := section {sections}
section := program_section | automata_section
program_section := [identifier] ":"  section_type statements
automata_section := [identifier] ":"  "automaton_section" states transitions
section_type := "start_section" | "priority_section"
  | "election_section"  | "task_activation_section" | ...

states := state {states}
transitions := transition {transitions}
state := identifier ["initial"] ":" "state" ";"
transition  := identifier "-->" "[" [guards],[clocks],
   [synchronizations] "]" "-->" identifier ";"
synchronizations : =synchronization {synchronizations}
synchronization := identifier '!'  | identifier "?"
clocks := assignment {clocks}
guards := expression

statements := statement {statements}
statement := put | assignment | declare | while | for | if | return | random
put := "put" "(" identifier [, expression] [, expression]")" ";"
declare := identifier ":"  data_type [ ":=" expression ] ";"
assignment := identifier ":=" expression  ";"
if := "if" expression "then" statements [  "else" statements ] "end" "if" ";"
return := "return" expression  ";"
for := "for" identifier "in" ranges "loop" statements "end" "loop" ";"
while := "while" expression "loop" statements "end" "loop" ";"
random :=  "uniform" "(" identifier "," expression ","
        expression  ")" ";"
  | "exponential" "(" identifier "," expression  ")" ";"

data_type := scalar_data_type
  | "array" "(" ranges ")" "of"  scalar_data_type
ranges := "tasks_range"
  | "buffers_range" |  "messages_range" ...
scalar_data_type := "boolean"  |  "integer"  |  "random" ...
operator := "and"  | "or"  | "mod" | "<" | ">" | "<=" | ">=" ...

expression :=  expression operator expression
  | "max_to_index"  "("  expression  ")"
  | "lcm"  "("  expression  "," expression ")"  | ...
```

Fig. 8. BNF grammar of the Cheddar programming language

13. Mimoune, M.E.H., Pierra, G., Ait-Ameur, Y.: An ontology-based approach for exchanging data between heterogeneous database systems. In: ICEIS 2003: Proceedings of the 5th International Conference On Enterprise Information Systems, Angers - France, École Supérieure d' Électronique de l' Ouest (2003)
14. Plantec, A., Singhoff, F.: Refactoring of an Ada 95 Library with a Meta CASE Tool, ACM Ada Letters journal. 26(3):61-70. Also published in the proceedings of the International ACM SIGAda Conference, Albuquerque, USA (2006)