

Refactoring the feasibility tests framework of Cheddar

Master thesis in Information and Communication Technology

PHAM VAN CHINH

Department of Information & Communication Technology
University of Science and Technology of Hanoi
Intake 2012-2014

Supervisor: Professor Frank Singhoff, Professor Alain Plantec
University of Western Brittany
Lab-STICC, CNRS, UMR 6285

Tutor: Professor Daniel Chillet
University of Rennes 1
ENSSAT

Abstract

This report describes a work in the research topic *Refactoring the feasibility tests framework of Cheddar* during the 6 months internship at Lab-STICC, CNRS, UMR 6285, University of Western Brittany, in the context of SMART project and Cheddar tool set. Cheddar is a tool set of verification the constraints of time for real-time systems. This simulator allows architecture designers to model a real-time system and to investigate its behaviour. In the work, there are three important achievements. The first achievement is to do a survey of many kinds of feasibility tests for mono-processor. The second achievement is to give a meta-model for automatic generation code Ada for feasibility tests. The third achievement is to implement the missed feasibility test into Cheddar by using method automatic generation.

The report is organized in five chapters. *Chapter 1 - Introduction* gives the context of the internship and the problem and motivation for the research . In *Chapter 2 - Preliminary definitions*, we revise all kinds of feasibility tests on mono-processor in real-time systems. *Chapter 3 - Meta-Model* presents the meta-model of feasibility test for method of automatic generation code. *Chapter 4 - Evaluation* give some experiments on generating code of feasibility tests and verification of code generated. The 5th and 6th chapters are *Related Works* and *Conclusion*.

Acknowledgements

Special thanks to professor Frank Singhoff and professor Alain Plantec and all colleagues in Lab-STICC, UBO. Thank you very much for all the supports not only in research activities but also in living conditions during my present in Lab-STICC. I would like to thank to all people in departement ICT who always give me supports during my 2 years at USTH.

Pham Van Chinh, Brest, France, 01/09/14

Contents

List of Figures

List of Tables

1

Introduction

Real-time systems are systems that have to meet time constraints. Cheddar is a toolset of verification constraints of time for real-time system. There are many kinds of feasibility tests which we want to integrate in Cheddar, each test has a formula to compute the constraint of time for a given task set.
In the current development of Cheddar, feasibility tests are manually programmed. Each test is implemented as an Ada function. As the project Cheddar evolves, more tests are implemented, more difficult to manage this part of Cheddar implementation. Another issue comes from the difficulties to reuse a test or a part of a test implementation to implement a new one. However, a test specification is rarely independent from another test that are already implemented.
We then propose an approach for automatic generation code of feasibility test. Automic code generation make the work of verification and validation easier and feasibility tests can be more faster implemented in the project. The code generated must be adaptive with the current development of cheddar and avoid of the duplication of tests which have been implemented.
At first, we present a taxonomy of feasibility test and then our meta-model of mathematical formula of feasibility test by language EXPRESS. Thirdly, we present how to generate the code Ada adaptive with the current development of Cheddar and finally, the evaluation code generated and related work in the end.

2

Preliminary definitions

In this chapter, we go to detail of feasibility test and introduce some kinds of feasibility test. We present in section 2.1 the feasibility test Preemptive Rate Monotonic, in subsection 2.2 the feasibility test Non Preemptive Rate Monotonic, in the section 2.3 the feasibility test Non Preemptive Earliest Deadline First, in the section 2.4 the feasibility test Demand on Processor (Any preemptive fixed priority scheduler), in the section 2.5 the feasibility test Non preemptive Rate Monotonic, in the section 2.6 the feasibility test preemptive Earliest Deadline First, in the section 2.7 some feasibility test base on worst case response times, in the section 2.8 the feasibility test any preemptive fixed priority scheduler, in the section 2.9 the feasibility test preemptive Deadline First.

2.1 Preemptive Rate Monotonic

Liu and Layland[7], in 1973, studied the fixed priority scheduling theory with some assumptions:

- all tasks are periodic
- all tasks are released at the beginning of period and deadline is equal to period
- all tasks are independent
- all tasks are released at time 0
- execution time is less than or equal to period ($C_i \leq D_i = P_i$)
- no task suspends itself
- all tasks are pre-emptive
- context switch has no cost
- there is only one mono-processor

All tasks is meet deadline when the following condition is satisfied:

$$U = \sum_{i}^{n} \frac{C_i}{P_i} \leq B \quad with \quad B = n(2^{\frac{1}{n}-1}) \tag{2.1}$$

U is factor of utilisation of processor. When n-{number of task} go to infinity, B is equal to ln(2).

2.2 Non preemptive Rate Monotonic

For the case there is no preemption, two tasks have different priority. The blocking time B_i is the waiting time for critical section (for example : sharing resources between tasks). For the case two tasks with different priority attempt to access data, if the task with higher priority get asses to resources first, there is no blocking time B_i, but if the task with lower priority get the asses to resources first, then the higher priority task request to access to resources. This task must wait for the release the shared data from the lower priority. The blocking time is consider an inversion of priority. We then must take into account the factor of blocking time B_i with period of current task P_i.
A set of periodic task using the priority ceiling protocol is scheduled if the following condition is satisfied[7]:

$$\forall i, 1 \leq i \leq n : \sum_{k=1}^{i-1} \frac{C_k}{P_k} + \frac{C_i + B_i}{P_i} \leq i(2^{\frac{1}{i}} - 1) \tag{2.2}$$

2.3 Non preemptive Earliest Deadline First

A periodic task set T=$\{T_1$,$T2$,..., $T_n\}$, for $i > j, P_i > P_j$ is scheduled if satisfy these conditions[9]:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \quad and \tag{2.3}$$

$$\forall i, 1 < i \leq n : C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{P_j} \right\rfloor . C_j \leq L \quad with \quad P_1 < L < P_i \tag{2.4}$$

Condition (2.3) is considered as the processor is not overload.
Condition (2.4) verifies if the upper bound of task T_i on the processor demand that can be realized in an interval of length L starting at the time release of a task T_i is scheduled. L denotes any constant of time between task T_1 and T_j

2.4 Any preemptive fixed priority scheduler

Consider a periodic task set T =$\{T_1$,T_2,T_3,..., $T_n\}$. Lui and Layland [7] define the cumulative demands on processor at time in [0,t] of tasks T_1,T_2,..,T_i[10]:

$$W_i(t) = \sum_{j=1}^{i} C_i \cdot \lceil \tfrac{t}{P_j} \rceil$$

Then if task T_i is scheduled if the condition below is respected :

$$\forall 1 \leq i \leq n : min_{0 \leq t \leq D_i} \left(\sum_{j=1}^{i} \frac{C_j}{t} \cdot \left\lceil \frac{t}{P_j} \right\rceil \right) \leq 1 \tag{2.5}$$

For task $T_j < T_i$, consider following definition :

- $L_i = \frac{W_i(t)}{t}$
- $S_i = \{kT_j | j = 1,2,..,i; k = 1,..,\lfloor \frac{T_i}{T_j} \rfloor\}$

The equation(2.5) becomes:

$$L_i = min_{\{t \in S_i\}} L_i \tag{2.6}$$

Remark: For automatic generation of code for (2.6) is possible because t is discrete and limited. But we must develop an algorithm for computing the minimum of expression for t $\in S_i$ and a function for computing S_i. Then, we must do operation comparison equal or less than 1.

2.5 Non preemptive Rate Monotonic

We suppose that :

- all tasks are periodic and independent.
- task set is concrete and synchronous.
- each task is released on deadline.
- task i-1 is less priority than task i.

This feasibility test is based on factor utilisation of processor. As there is no mechanism of preemption, we must take into account the maximum of factor blocking time B_i over the period P_i of all the task has higher priority than its current task $max_{1<i\leq n}\left(\frac{B_i}{P_i}\right)$. All tasks are schedulable if the following condition is satisfied [11]:

$$\forall i, 1 \leq i \leq n : \sum_{i=1}^{n} \frac{C_i}{P_i} + max_{1<i\leq n}\left(\frac{B_i}{P_i}\right) < n(2^{\frac{1}{n}} - 1) \tag{2.7}$$

2.6 Preemptive Earliest Deadline First

We suppose that :

- task set is synchronous and concrete
- all tasks are independent and periodic
- task T_i with deadline is greater than period $D_i > P_i$
- all task will be executed in a single processor

All tasks are scheduled if the following conditions are satisfied[12] [13]:

1. $U = \sum_{i}^{n} \frac{C_i}{P_i} \leq 1$
2. $\forall t \geq 0 : h(t) \leq t$

with

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right)$$

2.7 Tests based on worst case response times

Assumption for task set $T = \{T_1,T_2,..,T_n\}$ on single processor:

- each task has its fixed priority
- priority of tasks is sorted decreasing in the task set
- task set is concrete and synchronous
- all tasks are independent
- B_i is the blocking time of current task T_i
- J_i is the worst-case delay between a task arriving
- $hp(i)$ is task set with higher priority than task T_i

When the current task T_i is not blocking by any task with lower priority. The worst case response time of task T_i is computed by equation[14]:

$$r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil .C_j \tag{2.8}$$

When the current task T_i is spent B_i time blocking by a lower priority task, the worst case response time of T_i is computed by following equation[14]:

$$r_i = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil .C_j \tag{2.9}$$

When there's a jitter between two tasks i and i+1, the worst case response time of task T_i is computed by the following equation[14]:

$$r_i = C_i + J_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i + J_j}{P_j} \right\rceil .C_j \tag{2.10}$$

2.8 Any preemptive fixed priority scheduler

Assumption for task set $T = \{T_1,T_2,..,T_n\}$ on single processor:

- each task has its fixed priority
- priority of tasks is sorted decreasing in the task set
- task set is concrete and synchronous
- all tasks are independent
- B_i is the blocking time of current task T_i
- J_i is the worst-case delay between a task arriving

The worst case response time of each task T_i is computed by (2.11)[14] [15] [16]:

$$r_i = max_{q=1,2,..}(J_i + B_i + w_i(q) - q.P_i) \tag{2.11}$$

with

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{P_j} \right\rceil .C_j$$

and

$$\text{for q satisfies inequality: } w_i(q) \geq (q+1).P_i$$

2.9 Preemptive Earliest Deadline First

Suppose that a task set T=$\{T_1,T_2, ..,T_n\}$ is executed in a monoprocessor :

- task set is not concrete
- all tasks are independent
- priority is sorted decreasing after its deadline, so for j<i, task T_j is higher priority than task T_i

The worst case response time of task T_i is computed by the following formula[17] [18]:

$$r_i = max_{a \in S}(L_i(a) - a) \tag{2.12}$$

with :

$$L_i(a) = W(a,L_i(a)) + \left(a + \left\lfloor \frac{a}{T_i} \right\rfloor \right)$$

and

$$S = \bigcup_{j=1}^{n} \left(k.T_j + D_j - D_i, 0 \leq k \leq \lfloor \frac{min(\lambda, L_i)}{T_j} \rfloor \right)$$

$$\lambda = \sum_{j=1}^{n} \left\lceil \frac{\lambda}{T_j} \right\rceil \cdot C_j$$

With L_i we denote the length of the longest such deadline busy period for task T_i. L_i, for the non preemptive case, is calculated by the formula :

$$L_i = max_{j>i}(C_j - 1) + \sum_{j \leq i} \lceil \frac{L_i}{T_j} \rceil \cdot C_j$$

For the preemptive case, L_i is computed by the following equation :

$$L_i = \sum_{j \leq i} \lceil \frac{L_i}{T_j} \rceil \cdot C_j$$

And the higher priority workload :

$$W(a,t) = \sum_{D_j \leq a + D_i} min \left(\lceil \frac{t - S_j}{T_j} \rceil, 1 + \lfloor \frac{a + D_i - D_j}{T_j} \rfloor \right)$$

where $S_j = 0$ if $j \neq i$ and $S_j = a - \lfloor \frac{a}{T_i} \rfloor \cdot T_i$ else.

2.10 Conclusion

In conclusion, three kinds of feasibility tests are studied: feasibility tests based on factor of utilisation of processor, feasibility tests of demand on processor and feasibility tests by calculating the worst case response time. Each kind of tests has its assumptions and constraints to be respected. The feasibility tests by computing worst case response time is the most complicated and difficult to implement.

3

Meta-model of feasibility test formula

In the previous chapter, we have seen some kinds of feasibility tests for mono-processor. Now, we present in section 3.1 the meta-model of tool-set Cheddar. Section 3.2 presents an introduction of language EXPRESS. Section 3.3 presents global view of our meta-model. Section 3.4 presents the meta-model of generic symbol expression. Section 3.5 presents meta-model of an constant. Section 3.6 presents meta-model of an element of array. Section 3.7 presents meta-model of unary expression. Section 3.8 presents meta-model binary expression. Subsection 3.9 presents meta-model of binary comparison. Section 3.10 presents meta-model of binary Union. Section 3.11 presents meta-model of multi-ary expression.

3.1 Cheddar meta-model

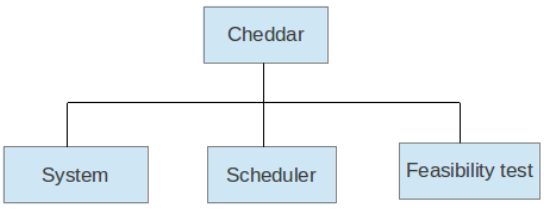

Figure 3.1: Cheddar meta-model

The structure of tool set Cheddar is represented in the Figure 3.1. Cheddar has 3 main meta models: the system, the scheduler and the feasibility test generated by tool Platypus based on STEP. In the system, there is hardware components and software components. The software component is depicted in the Figure 3.2 and it's included in meta-model of System. There are three kinds of software components:

- Messages are present mostly in the real-time systems. They can be periodic messages or aperiodic messages

- Buffer is a software component for storing data or resources. It has a specific address in memory
- A software component task can be periodic or aperiodic. Tasks are generated according to Poisson distribution with interarrival time $1/\lambda$. Each Poisson task can be a sporadic task, a parametric task, a scheduling task or frame task. They are connected to static memory components. The access to task passes through classical priority inheritance protocols such as PCP or others.

Objectif of the internship is to focus on the periodic tasks by analysing the schedulability of a given task set via formula of feasibility tests. Therefore, we give a meta-model of feasibility test for generating automatically the code source of feasibility test on mono-processor.

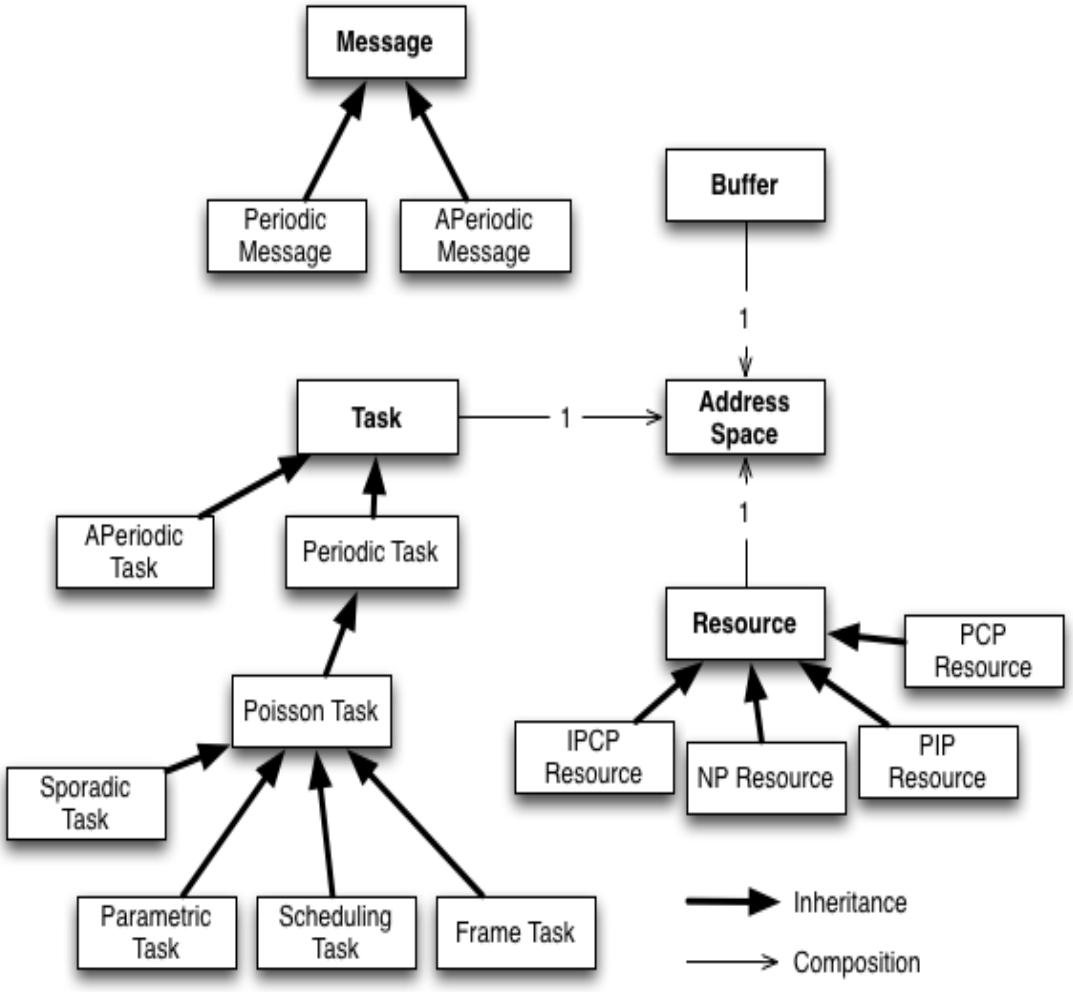

Figure 3.2: Cheddar main ADL software components

The meta-model of generated feasibility test is placed at feasibility test. The generation function test is added in this part.
Now, we describe our approach for modelling a mathematical formula of feasibility test. First, we analyse all the formula of feasibility tests. Then, we divide a mathematical formula into smaller elements that can be modelled. Here, we define all smaller elements, they can be:

- a constant
- element of an array: capacity C_i, deadline D_i, period P_i, blocking time B_i, jitter J_i.
- unary operation: Floor, Ceil.
- binary operation: addition, subtraction, multiplication, division, less operation, equal less operation and union operation.
- multi-ary operation: Sum, Max, Min.

We use language EXPRESS for modelling formula of feasibility tests. After analysing all formula of feasibility tests, we define all elements and necessary operations in the Figure 3.3

3.2 Express language

In this part, we present a brief introduction about Express language for understanding our meta-model.

- ENTITY: is a class
- SUBTYPE: is heritage of super class
- ABSTRACT: declare abstract class
- Instance: include an identification and a representation
- DERIVE: declare procedure or function of class
- There is no restriction on type of attribute
- Class can have some operations (or procedure)
- Type of variable can be enumeration, logical, boolean, binary, number, real, integer or string.

3.3 Hierarchy of expression language

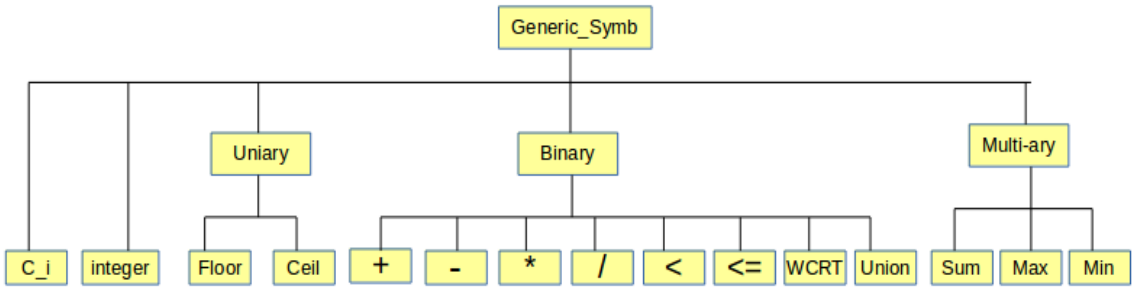

Figure 3.3: Hierarchy of expression language

We present our global view of our expression language in the Figure 3.3. In the root, we declare a Generic symbol, it can be any formula of feasibility test. From the left to the right, there are a constant and an element of array derive from the root. A constant is present in many feasibility

test, it is considered as time addition taken in account to the formula of feasibility test. An element of array can be an capacity, deadline, period, jitter, blocking time.
Expression uniary derived from the root, it's meta-model for operation floor and operation ceil.
Expression binary concludes all operation addition, subtraction, multiplication, division, comparison or union. They are derived from the root . Expression binary has 8 sub-classes. We consider operation comparison less than or equal and less than as binary operation, but its code generations is different from others binary expression. Expression WCRT is added for calculating the worst case response time of a task set.
Expression multiary consists of 3 symbols Sum, Max and Min.

3.4 Generic Symbol Expression

It presents the root of our expression language. It can be itself a formula. We define it as an abstract class.

```
ENTITY Generic_Symb_Expression
 ABSTRACT SUPERTYPE;
 operator_type  : Symb_Operator_Type;
   DERIVE
   code_latex   : STRING
   code_ada     : STRING
   code_ada_WCRT : STRING
END_ENTITY;
```

Its attributes code_latex, code_ada and code_ada_WCRT are defined as string for generation automatic code.

3.5 A Constant

A Constant on the formula represents an integer. It is defined as a symbol with attribute string. It's a subclass of entity Generic_Symbol_Expression.

```
ENTITY A_Constant_Expression
  SUBTYPE OF (Generic_Symb_Expression);
  An_Element : STRING;
END_ENTITY;
```

3.6 An element of array

An element of array can be capacity C_i, deadline D_i, period P_i, blocking time B_i, jitter J_i or others. It presents a value of array integer at index i. It is defined by entity An_Element_Array_Expression with two parameters : index and a main element. The main element can be C_i, P_i, B_i, J_i, C_j, P_j, B_j or J_j. The i represents the current task, and the j represents the task with higher priority.

```
ENTITY An_Element_Array_Expression
  SUBTYPE OF (Generic_Symb_Expression);
  Index : STRING;
  Element : Symb_Element_Array_Type;
END_ENTITY;
```

3.7 Unary expression

The Unary expression is derived from Generic_Symbol_Expression. It's the super class of operation Floor and operation Ceil. An Unary expression contains a generic expression and a type of operation (floor or ceil). VALUE attribute is a generic expression, it can be a complex formula.

```
ENTITY Symb_Unary_Expression
  SUBTYPE OF (Generic_Symb_Expression);
  VALUE : Generic_Symb_Expression;
  operator : Symb_Operator_Type;
END_ENTITY;
```

3.8 Binary expression

The binary expression is derived from Generic_Symbol_Expression. It's the supper class of operation plus, subtraction, multiplication, division which has 2 Generic_Symbol_Expression, one on left value and one on right value.

```
ENTITY Symb_Binary_Expression
  SUBTYPE OF (Generic_Symb_Expression);
  rvalue : Generic_Symb_Expression;
  lvalue : Generic_Symb_Expression;
END_ENTITY;
```

For modelling all binary expressions, we need to declare another sub class for each operation: addition, subtraction, multiplication, division or union. For example, expression $A + B$ with A, B a constant, or a Generic_Symbol_Expression. We declare another subclass Symb_Plus_Expression of superclass Symb_Binary_Expression.

```
ENTITY Symb_Plus_Expression
  SUBTYPE OF ( Symb_Binary_Expression );
  DERIVE
    (function for generating code)
END_ENTITY;
```

3.9 Binary comparison

Binary comparison is a special binary expression (operation less than and operation equal and less than). It derives from class abstract Generic_Symb_Logic_Expression. It contains 2 main attributes rvalue and lvalue. rlvalue is a Generic_Symbol_Expression, and rvalue is a constant.

```
ENTITY Generic_Symb_Logic_Expression
 ABSTRACT SUPERTYPE;
 operator_type : Symb_Logic_Type;
 rvalue : Generic_Symb_Expression;
 lvalue : Generic_Symb_Expression;
DERIVE
       (function for code generation)
END_ENTITY;
```

3.10 Binary Union

We denote operation Union as a binary expression (4.1). It derives from the class Binary expression. It has 2 values, one for computing the value, one for condition for computing the value.

```
ENTITY Symb_Union_Expression
       SUBTYPE OF ( Symb_Binary_Expression );
       DERIVE
              (function for code generation)
END_ENTITY;
```

3.11 Multi-ary

The multi-ary expression is a subclass of Generic_Symbol_Expression. Its attributes are index, lower bound, and a formula (Generic_Symbol_Expression). It's the super class of sub class Sum and class Union. A Multi-ary expression represents for symbol Sum or Min, Max of an array integer. A Sum of a Generic_Symbol_Expression is a loop with index go from lower bound to upper bound.

```
ENTITY Symb_Multiary_Expression
  SUBTYPE OF ( Generic_Symb_Expression );
  index : STRING;
  lower_bound : STRING;
  formula : Generic_Symb_Expression;
  symb_expression : Symb_Operator_Type;
END_ENTITY;
```

3.12 Conclusion

In conclusion, an approach by using meta-model for modelling mathematical formula of feasibility tests is presented. We have seen also how to model feasibility tests using language EXPRESS. Each feasibility test is composed of smaller elements which is modelled. Each small elements is presented by its meta-model.

4

Generation code Ada from meta-model

In the previous Chapter, we presented our meta-model of symbolic language for modelling the formula of feasibility test. Now, by using meta-model of mathematical formula, we assemble from smaller elements to form a formula of feasibility test. Each formula is composed of many basic elements. We use a function to generate Ada code which adapts with the current development of Cheddar. We firstly analyse the code source of Cheddar. Then, we try to a generate each small elements from the symbolic language to Ada.

4.1 A constant

In many feasibility tests, there are some constants of time in the formula. For example, a L symbol. The code generated for a Constant is itself. It's a integer variable.

4.2 An element of an array

An element of an array is present in all feasibility tests. We model all elements concerning for one task. One task can have a capacity, a deadline, a period, a blocking time, a jitter, an offset and a priority. A capacity C_i is time of execution of a task T_i. Deadline D_i is the time of task T_i must be completed before. A period T_i of task i is time between each iteration of regularly repeated task. A Jitter J_i is the delay between the invocation of a task T_i and its release (when it actually starts to execute). Blocking time B_i is waiting time in critical section of higher priority T_i blocked by the lower priority task.
We define that the current task has the capacity C_i, deadline D_i, period P_i, jitter J_i, blocking time B_i . The task with higher priority of current task has period C_j, deadline D_j, period P_j, jitter J_j, and Blocking time B_j. In the Cheddar, we make some concept of element of a task set. First, we find where the pointer points to the ordered task set with function *current_element (My_Tasks, Taskj, Iterator)* and then we compare the priority of the pointer with the priority of current task. Its costs (n-i) comparisons with n - number of tasks and i- the i^{th} task of the ordered task set. So

we denote elements of the task with higher priority :

- Taskj.capacity: capacity of the higher priority task
- Periodic_Task_Ptr (Taskj).deadline: dealine of the task with higher priority
- Periodic_Task_Ptr(Taskj).period: the priority of task with higher priority
- Periodic_Task_Ptr (Taskj).jitter: the jitter of task with higher priority
- Periodic_Task_Ptr (Taskj).blocking_time: the blocking time of task with higher priority.

	Task with higher priority
Capacity	Taskj.capacity
Deadline	Periodic_Task_Ptr (Taskj).deadline
Period	Periodic_Task_Ptr(Taskj).period
Jitter	Periodic_Task_Ptr (Taskj).jitter
Blocking time	Periodic_Task_Ptr (Taskj).blocking_time

Table 4.1: Code generation for element of array of task with higher priority

For the current task, first we initialize the task by using this code *current_element (My_Tasks, Taski, Iterator2)* and then we use a pointer to point to this current task. The code generation for the elements of current task is showed in table 4.2.

	Current task
Capacity	Current_Task.capacity
Deadline	Periodic_Task_Ptr (Taski).deadline
Period	Periodic_Task_Ptr(Taski).period
Jitter	Periodic_Task_Ptr (Taski).jitter
Blocking time	Periodic_Task_Ptr (Taski).blocking_time

Table 4.2: Code generation for element of array of current task

As the current development of Cheddar, we consider time as an integer for simulating. For the code generation, we need to put a converter type Double before each element of array generated. For example Double(Current_Task.capacity), Double(Periodic_Task_Ptr (Taskj).capacity), ... The code Ada is generated by a function returning a String.

4.3 Code generation for uni-ary expression

Uni-ary expression has only 2 operations Floor and Ceil. The current development of cheddar has integrated these two operations. The generation code is below :

- Floor operation : Double'Floor(EXP.VALUE.code_ada)

- Ceil operation :Double'Ceil(EXP.VALUE.code_ada)

EXP : is a generic formula
code_ada : is a function that generates the formula and return the formula as a string. We call a function for generating code of operation Floor function Symb_Floor_Expression_code_ada (EXP : Symb_Ceil_Expression) : STRING. For the operation Ceil, we use another function for generation code : function Symb_Ceil_Expression_code_ada

4.4 Code generation for binary expression

By analysing all the feasibility tests introduced in the chapter 2, we define all necessary binary operations of a feasibiity tests is:

- Addition
- Subtraction
- Multiplication
- Division
- Comparison (less than and equal or less than)
- Union expression

In the meta-model of binary expression, there are two attributes : lvalue and rvalue. Each value is an generic expression. An generic expression can be itself a formula. Because there are two kinds of computation for the feasibility test:

- compute without depending on the previous result
- compute with the previous result

Therefore, we define 2 others functions generating code:

- computing a general formula
- computing the value of worst case response time.

4.4.1 Addition

There are two kinds of binary expression Addition :

- for computing general formula:

$$C_i + B_i$$

- for computing worst case response time:

$$C_i + \sum_{\forall j \in hp(i)} \lceil \frac{r_i}{P_j} \rceil \cdot C_j$$

Here, the generation code for a general formula is $lvalue + rvalue$ with lvalue is C_i and $rvalue$ is $C_i + B_i$. For the second binary Addition, there is a multiary expression on the right value. The generated function must have a loop for summing up $\lceil \frac{r_i}{P_j} \rceil \cdot C_i$ of all tasks which have higher priority than the current task T_i.

4.4.2 Subtraction

Operation subtraction is considered as an operation between 2 constants of time (or 2 integers). Code generation for this binary expression is $lvalue - rvalue$ which lvalue and rvalue are generic expression. It's forbidden to have an multi-ary expression either on left value or on right value.

4.4.3 Multiplication

The same as the operation addition, we have two values $lvalue$ and $rvalue$, each is a generic expression. Code generation of this operation is $lvalue * rvalue$. It's possible to have a multiary expression on the left value or on the right value.

4.4.4 Division

The code generation of this operation is $\frac{lvalue}{rvalue}$ which lvalue and rvalue are generic expression. It's not allowed to have any multi-ary expression either on left value $lvalue$ or on right value $rvalue$.

4.4.5 Comparison

In formula of feasibility test, there are two kinds of comparison : less than and equal or less than. The code generation for this comparison is more complex. First, we generate the code of function for computing the left value lvalue, then another function call this first function result to compare with the right value $rvalue$. The result of this comparison returns a boolean.

4.4.6 Binary WCRT

It's added for computing some formula like (2.8). We must take into account the existent of right value which is a multi-ary expression. So, the generation code for binary WCRT formula is more complex. In the generated function, there are 2 loops. One inner loop sums up all values $\lceil \frac{r_i}{P_j} \rceil . C_j$ of all tasks which have higher priority than current task. One outer loop computes the next value r_i^k in function of r_k^{k-1} . The condition for existing the outer loop is the previous computation is equal to the current computation ($r_i^k = r_i^{k-1}$) or the current computation is bigger than the period of current task T_i ($r_i^k > T_i$). It's forbidden to have an multi-ary expression on left value. The multi-ary expression is present only on the right value. The complexity of computation of this equation is $\theta = n * m$ with n is the number of task, and m is number of iteration on the inner loop. The complexity of computing the WCRT is order of $\theta = n^2$.

4.4.7 Union Expression

Consider formula of an Union expression below :

$$S = \bigcup_{j=1}^{n} \left(k.T_j + D_j - D_i, 0 \leq k \leq \lfloor \frac{min(\lambda, L_i)}{T_j} \rfloor \right) \tag{4.1}$$

The code generation for Union expression like (4.1) is a function with 2 loops for computing all values of set S. One outer loop varies j from 1 to n. One inner loop computes $lvalue = k.T_j + D_j - D_i$ for all values k satisfies condition of $rvalue$: $0 \leq k \leq \lfloor \frac{min(\lambda, L_i)}{T_j} \rfloor$.

4.5 Generation code for multiary expression

Code generation of multiary expression is more complex. It can be many functions with loops for computing the the formula of feasibility test. And the code generation must be adaptive with the current development of Cheddar. We try to reuse written code of Cheddar for generating code from our meta-model.

4.5.1 Code generation for expression sum

Consider formula processor utilisation factor (2.1). For this example, we generate first function for computing processor utilisation factor ($lvalue$). Condition for exiting the loop of the first function is at the end of task list. A second function for the comparison the $lvalue$ with the constant B($rvalue$). The result of second function returns a boolean.

4.5.2 Generation code for function Min, Max

Consider the (2.12), the code generation for this kind of formula is 2 functions. One function computes an array value of $L_i(a)$ with $a \in S$. And one another function with a loop computes the maximum of $L_i(a) - a$. The algorithm for finding the maximum is described below :

```
max := 0.0;
for j from 0 to nb_element of array S
   temp:=L_i(S(j))-S(j);
   if (max < temp) then
   max := temp;
   end if
end for;
return max;
```

For computing S(j), we use meta-model of Union Expression which is described from section Binary Expression.

4.6 Structure of function generated

We generate one or many functions for computing the formula. Structure of a function generated Figure 4.1 is divided into 3 parts. The first part is the beginning of the program. The second part is the loop with the generated formula . The third part is the end of function. In the beginning of the function, we put the part of declaration of input variables or input data, and the initiation of local variables. In the middle of function, the loop will be generated for computation. The last part, we return the result of computation.

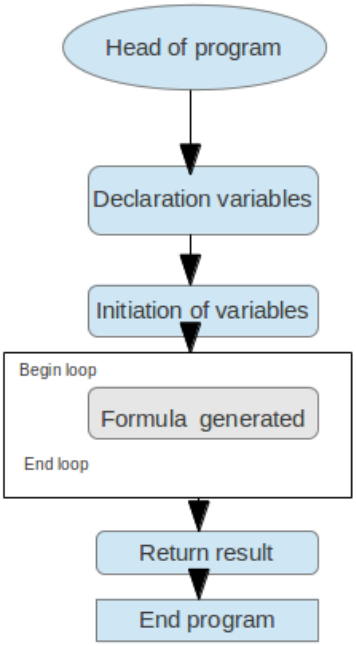

Figure 4.1: Structure of function generated

4.7 Conclusion

In conclusion, the automatic generation code Ada for feasibility tests is presented by its smaller elements. Each smaller element has its own code generation. Therefore, the complex formula of feasibility tests which is composed by many smaller elements, can be easier to be generated. However, The code generation for multiary expression can be many functions with loops. An analysis of the code of Cheddar is also studied so that code generation can be adaptive with current development of project Cheddar.

5

Evaluation

In previous chapter, the automatic generation Ada code for smaller elements of formula of feasibility tests is presented. Now, we do evaluation the code generated by choosing some formulas from which the function generation will be used for producing the code. Firstly, we want to generate code latex of the formula for being sure that our meta-model is correct. Then, we generate code Ada and integrate this code on Cheddar, and verify if it gives the expected result. In this chapter, we also see how to use meta-model to build a function for producing code of the chosen feasibility test.

5.1 Formula construction

For building a formula, we need to use the small elements that we defined above. Step by step, we call its instance for putting together these small elements to build the wanted formula.

- For creating a constant, we call a instance of a_constant_expression. For creating a constant we call its expression. For example, we need a constant L dans le formula, just call a new instance : A_CONSTANT_EXPRESSION(.PLUS_TYPE.,'L')
- For creating an element of array, we call an instance of an_element_array_expression. For example, if we need a capacity of current task, we call its expression : AN_ELEMENT_ARRAY_EXPRESSION(.PLUS_TYPE., 'i', .C_I.)
- For creating a uniary expression, for example a Ceil operation, we need to call an instance of ceil operation symb_ceil_expression. In this example #7 is a formula which the operation will compute this expression. SYMB_CEIL_EXPRESSION(*, #7, .PLUS_TYPE.)
- For creating a binary expression, we call instance of binary expression. For example the addition of 2 generic expressions SYMB_PLUS_EXPRESSION(*, #5, #1);
- For creating an multiary expression,in this case, the sum of a formula : SYMB_SUM_EXPRESSION(.PLUS_TYPE., 'j', 'n', #8, *). #8 is a generic expression.

We go to the details of an example:

5.2 Experiment 1

Consider the formula (2.8) that we want to model. With its formula, we want to compute all the worst case response time (WCRT) of each task. By using the meta-model, we need to create the formula steps by steps which is described below:
(*declare C_i*)
#1=AN_ELEMENT_ARRAY_EXPRESSION(*,'i',.C_I.)
(*declare C_j*)
#2=AN_ELEMENT_ARRAY_EXPRESSION(*,'j',.C_J.)
(*declare P_j*)
#3=AN_ELEMENT_ARRAY_EXPRESSION(*,'',.P_J.)
(*declare a constant*)
#4=A_CONSTANT_EXPRESSION(*,'R_i')
(*ceil(R_i/P_j)*)
#5=SYMB_CEIL_EXPRESSION(*,#6,.PLUS_TYPE.)
(*R_i/P_j*)
#6=SYMB_DIVIDE_EXPRESSION(*,#3,#4)
(*ceil(R_i/P_j)*C_j*)
#7=SYMB_MULTIPLY_EXPRESSION(*,#5,#2)
(*sum of ceil(R_i/P_j)*C_j*)
#8=SYMB_SUM_EXPRESSION(*,'j','n',#7,*)
(*C_i + sum of ceil(R_i/P_j)*C_j*)
#9=SYMB_SUM_WCRT_EXPRESSION(*,#8,#1)

After do all those steps, we formed the formula as we wish. We now take a task set to test with our code generated. We then generate the code of feasibility test $r_i = C_i + \sum_{\forall j \in hp(i)} \lceil \frac{r_i}{P_j} \rceil . C_j$

```
function compute
(My_Tasks    : in Tasks_Set;
Current_Task : in Generic_Task_Ptr)
return       Double
is
Iterator, Iterator2 : Tasks_Iterator;
Taskj,Taski   : Generic_Task_Ptr;
calcul, tmp  : Double;
begin

calcul := 0.0;
tmp:=-0.1;
current_element (My_Tasks, Taski, Iterator2);
While (tmp/=calcul) loop
 reset_iterator (My_Tasks, Iterator);
 tmp :=calcul;

 calcul := Double(Taski.capacity);
 loop
```

```
  current_element (My_Tasks, Taskj, Iterator);
  if (Taskj.priority > Current_Task.priority) then
    calcul := calcul +
    Double(Taskj.capacity)*
    Double'Ceiling((tmp/Double(Periodic_Task_Ptr (Taskj).period)));
  end if;
  exit when is_last_element (My_Tasks, Iterator);
  next_element (My_Tasks, Iterator);
end loop;
end loop;
return calcul;
end compute;
```

Consider a task set with 3 tasks T1, T2, T3. By theory, we obtain $R_1 = 3$, R_2=5 and R_3=18.

Task name	Capacity	Period
T1	3	7
T2	2	12
T3	5	20

Table 5.1: Task set example

The code generated gives us a function named *compute* which produces the same result as above. Hence, we validate our code generation and our method to generate automatic code.

5.3 Experiment 2

Consider the formula we want to evaluate for the test based on processor utilisation factor (2.3). Consider a same task set with 3 tasks T1, T2, T3. We then generate the code of function of feasi-

Task name	Capacity	Period
T1	3	7
T2	2	12
T3	5	20

Table 5.2: Task set example

bility test U $=\sum_{i}^{n} \frac{C_i}{P_i} \leq 1$ First, we generate function for calculating the right value of expression.

```
function compute
(My_Tasks      : in Tasks_Set;
Processor_Name : in Unbounded_String)
return          Double
is
```

```
  Taski    : Generic_Task_Ptr;
  My_Iterator : Tasks_Iterator;
  calcul : Double := 0.0;
begin
  Periodic_Control (My_Tasks, Processor_Name);
  reset_iterator (My_Tasks, My_Iterator);
  loop
  current_element (My_Tasks, Taski, My_Iterator);
  if (Taski.task_type /= Aperiodic_Type) then
    calcul := calcul +
    (Taski.capacity)/Double(Periodic_Task_Ptr(Taski).period);
  end if;
  exit when is_last_element (My_Tasks, My_Iterator);
  next_element (My_Tasks, My_Iterator);
  end loop;
return calcul;
end compute;
```

Second, we compare it with the right value in another function and return the result.

```
function compute_less_equal
 (My_Tasks      : in Tasks_Set;
  Processor_Name : in Unbounded_String)
  return boolean
  is
  rightvalue, leftvalue : Double;
  begin
  leftvalue :=compute(My_Tasks,To_Unbounded_String("CPU_A")) ;
  rightvalue := 1.0;
  if (leftvalue <= rightvalue) then
       return True;
  else return False;
  end if;
end compute_less
```

By theory, the factor of processor utilisation for this case is U = 0.845. For this comparison with constant 1 the function will return TRUE as the result. Generating the code source for computing this formula, we obtain the same result as theory.

5.4 Experiment 3

In this experiment, we generate code Ada for feasibility test (2.11). We first generate code for function W_i:

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{P_j} \right\rceil .C_j \tag{5.1}$$

$$W_i(q) \geq (q+1).P_i \tag{5.2}$$

And then we compare $W_i(q)$ with $(q+1).P_i$. For each q go from 0 to n, we compute $W_i(q)$. If $W_i(q)$ satisfy (5.2), we then store the value of $W_i(q)$ and increase k by 1. If condition (5.2) is not satisfied, we go out the loop and return a table of value $W_i(q)$.

```
procedure compute_less_equal
(My_Tasks     : in Tasks_Set;
Processor_Name : in Unbounded_String;
Current_Task : in Generic_Task_Ptr;
Value_W_i : out MY_ARRAY;
q: out integer)
is
  My_Iterator : Tasks_Iterator;
  Taski    : Generic_Task_Ptr;
  rightvalue : Double;
  leftvalue : Double;
  k: integer:=0;
begin
reset_iterator (My_Tasks, My_Iterator);
current_element (My_Tasks, Taski, My_Iterator);
rightvalue := 0.0;
leftvalue :=0.0;
Value_W_i(0):=1.0;
While (Value_W_i(k) >= rightvalue) loop
  leftvalue :=Double(W_i(My_Tasks,Double(k),Taski)) ;
  rightvalue :=Double(Periodic_Task_Ptr(Taski).period)
  *Double(1.0+Double(k));
  Value_W_i(k):= leftvalue;
  k:=k+1;
end loop;
q:=k;
end compute_less_equal;
```

We generate the function for computing $W_i(q)$ by our meta-model. The function is very similar to (2.8) which is calculated in the same algorithm in experiment 1. The iteration stops when $W_i^n(q) = W_i^{n-1}(q)$ or $W_i^n > P_i$ with precondition $W_i^0 = 0$.

```
function W_i
(My_Tasks    : in Tasks_Set;
q:in Double;
Current_Task : in Generic_Task_Ptr
)
return    Double
is
Iterator : Tasks_Iterator;
Taski, Taskj   : Generic_Task_Ptr;
calcul, tmp  : Double;
begin
calcul := 0.0;
```

```
tmp:=-0.1;
current_element (My_Tasks, Taski, Iterator);
While (tmp/=calcul) loop
 reset_iterator (My_Tasks, Iterator);
 tmp :=calcul;
  calcul := (Double(Current_Task.capacity))*(1.0+q);
   loop
     current_element (My_Tasks, Taskj, Iterator);
     if (Taskj.priority > Current_Task.priority) then
       calcul := calcul + double((Double(Taskj.capacity))*
       (Double'Ceiling((Double(Periodic_Task_Ptr (Taskj).jitter)+tmp)/
       (Double(Periodic_Task_Ptr (Taskj).period)))));
      end if;
    exit when is_last_element (My_Tasks, Iterator);
    next_element (My_Tasks, Iterator);
   end loop;
end loop;
return calcul;
end W_i;
```

For computing (2.11), we generate a function to compute the maximun of array. First, we compute all value r_i, then find the maximum of array r_i.

```
function max_r_i
(My_Tasks    : in Tasks_Set;
W_i          : in MY_ARRAY;
nb_value_W_i : in integer;
Current_Task : in Generic_Task_Ptr
)
return Double
is
My_Iterator : Tasks_Iterator;
Taski    : Generic_Task_Ptr;
q : integer;
calcul,max : Double;
begin
  calcul := 0.0;
  current_element (My_Tasks, Taski, My_Iterator);
  max :=0.0;
  for q in 0..nb_value_W_i loop
    calcul := Double(Periodic_Task_Ptr(Taski).jitter)+
    Double(Periodic_Task_Ptr(Taski).blocking_time) +
     W_i(q)-Double(q)*Double(Periodic_Task_Ptr(Taski).period);
  if calcul > max then
   max:=calcul;
  end if;
end loop;
return max;
end max_r_i;
```

5.5 Conclusion

In conclusion, for three cases studied, code generation gives us the same results with the code programmed by hand. The code generation can be produced any feasibility tests presented in the chapter 2. However, feasibility tests are generated into many functions or procedures which are needed for each complex formula of feasibility test.

6

Related works

In previous chapter, we have seen the evaluation of code generation, now we present the related work of modelling real-time system and feasibility tests using calculus equation and compare with our works.

ModelicaML[4] is a object-oriented modelling language. It is is tool set for modelling industrial environment. ModelicaUML is a UML profile. It gives the possibility to model system using meta-model with code implemented to control system. The function of calculus formula is integrated in the model for doing the control. They use formula for modelling the control system with diagram, the connection diagram, and for graphical modelling.

AADL[5] based on the synchronous language SIGNAL is used for modelling the system by blocks and intern signal and event signal. For based-time scheduling, it uses a trigger to activate each task. For computing worst case response time before sending to the simulator, it uses a *library cost function* to calculate the cost time of thread which are tested. The formula for time computing is described block by block. The execution of threads(or tasks), the storage of data and code, and the communication platforms are supported by execution platform components. SIGNAL allows the specification of multi-clocked systems for seeing if any signal is present in components.

HybridUML[3] for modelling system with meta-model using a part of expression which is added in the model for computing : AlgebraicExpression, DifferentialExpression and InvariantExpression. Gautier, Talpin proposed a notation of state machine for describing the system. Differential expression, algebraic expression are needed to use for describing analogue-real variable. They use CHARON and its basic mechanisms for modelling the hybrid system.

Gilles and Hugues developped REAL, an AADLv2 annex language[6] for modelling complex system with formula for computing processor utilization factor and generating code from the meta model with OCARINA.

O. Sokolsky, I. Lee [19] used ACRS for modelling real-time system for verification of constraints of time in mono processor with 2 variables e and t. e represents execution time of a thread and t is the elapsed time since its dispatch which some dispatch policy of threads in the system. But the computation of time on ACSR model is calculated in hardware components. The output of data on a connection presents its completion of execution on CPU.

A. Amano, M. Kawabata[20] proposed a method for generation code of simulation in any language for solving ODE (Ordinary Differential Equation). They use a TecML (Time Evolution Calculation

Markup Language) file for describing the input and output of a loop of calculation ODE and a CellML file for describing the mathematical description of biological models. RelML (Relation Markup Language) a language for describing the correspondence between variables in the CellML model file and the variable types in an ODE numerical solution or a coupling calculation scheme described in the TecML file. All the inputs, outputs, preconditions and conditions for exit from the loops must be specified for work of generation code. The method is very complex, not easy to manipulate for each simulation. More than that, there are too much parameters needed for the inner loop and outer loop for example: re, $initial$, $inner$, $loopcondition$, $final$ and $post$.

F.R. Punzalan, Y. Yamashita[21] introduce CellML a description language for generating executable file for simulation of ODE (Ordinary Differential Equation) which is the same method of Amano[20].

Our work is to use meta-model with language EXPRESS for generating the function of feasibility tests and then integrate the missed feasibility tests to source of Cheddar[2]. Moreover, we can not generate totally function of feasibility test, but for dividing the formula into enough small that can be generated automatically. The advantage of our method is to give a simple approach to generate code of each small element of mathematical formula of feasibility test and that makes us easy to verify the long programme of feasibility test. The work for code generation is different from others who have integrated the calculus function in their model for do the task. There is no meta-model or model using AADL, UML,... that was used to generate the code source for integrated in the system electronics or embedded system. The work of A. Amano, M. Kawabata [20] is very closed to ours but it doesn't solve the same problem. Our method is more simple and easier to manipulate.

7

Conclusion

In the report, we presented first a survey of feasibility tests and then an approach to model formula of feasibility tests using language EXPRESS. We describe how to produce generic code for any type of feasibility test from meta-model and show the possibility for generating automatically code for feasibility tests. Modelling formula of feasibility test makes our project easier to verify and validate if the function of feasibility test is correct. Using automatic generation code, the project evolves more rapid and simplify the work of testing and verifying code source for future development, and get out of the coincidence of feasibility with another. That means we avoid of duplicating some feasibility tests.

In the future, we want to model all kind of feasibility test with our meta-model for multi-processor feasibility test. The generation of code more and more adaptive with the software Cheddar for solving the problem of programming by "Ad-hoc".

Bibliography

[1] F. Singhoff, J. Legrand, L. Nana and L. Marcé Cheddar : a Flexible Real Time Scheduling Framework. ACM SIGAda Ada Letters, volume 24, number 4, pages 1-8. Edited by ACM Press, New York, USA. December 2004, ISSN:1094-3641.

[2] F. Singhoff. Investigating the usability of real-time scheduling theory with Cheddar project. Habilitation à diriger des recherches de l'Université de Bretagne Occidentale. Novembre 2008.

[3] K. Berkenkotter, S. Bisanz, U. Hannemann. HybridUML Profile for UML 2.0

[4] W.Schamai. Application of Model Based specification Approach in a Industrial Environment

[5] Y. Ma, H. Yu, T. Gautier, J-P. Talpin, L. Besnard, P. Le Gueni. System synthesis from AADL using Polychrony.

[6] O.Gilles, J.Hugues. Expressing and enforcing user-defined constraints of AADL models. In: Proceedings of the 5th UML and AADL Workshop (UML and AADL 2010)

[7] Lui and Layland. Scheduling Algorithms for multi-programming in a Hard Real-Time Enviroment. Journal of the Association for Computing Machinery,20(1):46-61, January 1973.

[8] L. Sha, R. Rajkumar, ans J.P Lehoczky. Priority Inherence Protocols: An Approach to real-time Synchronization. IEEE Transactions on computers, 39(9):1175-1185, 1990.

[9] K. Jeffay, D. Stanat, and C.Martel On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. in Proc. Of the IEEE Real Time Symposium (RTSS'91), San Antonio, Texas, December 1991.

[10] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotoic Scheduling Algorithm : Exact Characterization and Average Case Behaviour.page 166-171. In Proc. Of the IEEE Real Time System symposium (RTSS'89), December 1989.

[11] F.V.Carcahlo. Sur l'intégration de Mécanismes d'Ordonnancement et de Communication dans la sous-Couche MAC de Réseaux Locaux Temps Réel. Thèse de l'Université de Toulouse 3,1996.

[12] S.K. Baruah, R.R. Howell, and L. E. Rosier. Algorithms ans Complexity Concerning the Preemptive Scheduling of Periodic Real Time Tasks on one Processor. Real Time Systems journal 2: 301-324, 1990

[13] L. George, N. Di Natale. Minimizing memory utilisation of real time task sets in single and multi-processor system on chip, volume 3, pages 67-99. In Proceedings of the 22nd IEEE Real Time Systems Symposium, March 2001.

[14] M. Joseph ans P. Pandya. Finding Response Time in a Real-Time System. Computer Journal, 29(5): 390-395, 1996.

[15] A.N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority preemptive scheduling. Software Engineering Journal, pages 284-292, 1993.

[16] K.W. Tindell ans J. Clark. Holistic schedulability analysis for distributed hard real time systems. Microprocessing ans Microprogramming, 40(2-3):117-134, April 1994.

[17] L. George, N. Riverre, and M. Spuri. Preemptive Scheduling of periodic ans Sporadic Tasks. INRIA Technical report number 2966, 1996.

[18] Marco Spuri. Analysis of deadline scheduled real-time systems.Technical Report RR-2772, 1996.

[19] O. Sokolsky, I. Lee, D. Clarke. Schedulability Analysis of AADL Models

[20] A. Amano, M. Kawabata. A program code generator for multiphisics biological simulation using Markup Language.

[21] F.R. Punzalan , Y. Yamashita, N. Soejima, M. Kawabata,T. Shimayoshi, H. Kuwabara, Y. Kunieda and A.Amano. A CellML simulation compiler and code generator using ODE solving schemes.

[22] F Singhoff, A Plantec. Towards User-Level extensibility of an Ada library: an experiment with Cheddar. Reliable Software Technologies–Ada Europe 2007, 180-191

Appendices

A

Appendix A: Programming Tools and Ada Implementation

A.1 Programming Tools

All toolsets have been used for my work during the internship at Lab-STIC-UBO.

A.1.1 Platypus

Platypus is a STEP-based meta-environment. The tool is used for meta-models and code generation. The method of generation of code is the same in this article [22]. The method is used in my work based on language EXPRESS and toolset Platypus. We decribe the meta-model of feasibility test on EXPRESS and generate the code from meta-model with Platypus. Platypus run on the Pharo - an open-source Smalltalk-inspired environment.

A.1.2 The GNAT Programming Studio

GNAT Programming Studio (GPS) is an integrated development environment for Ada developed by AdaCore. The tool is available for download at:

http://libre.adacore.com/tools/gps/

A.2 Ada Implementation

The implementation of the work in the project, at the moment the report is written, can be found at the address below. This is integrated inside the principal branch of the Cheddar repository.

http://beru.univ-brest.fr/svn/CHEDDAR/trunk/src

B

Appendix B: Publication

This is the paper presented my work during the internship at Lab-STIC of UBO.

- *Meta-model for automatic generation Ada code for feasibility test.* V.C. Pham, F. Singhoff, A. Plantec. To be submitted at ACM Ada Letters Journal.