ARIMA

EPI sciences

Multi-target synthesis of logic controllers using a MDE approach

Gérard NZEBOP NDENOKA[*1,4,5], **Maurice TCHUENTE**[2,4,5], **Emmanuel SIMEU**[3,5], **Valery MONTHE**[2]

[1]Department of Land Surveying, National Advanced School of Public Works,
[−]P.O. Box 510 Yaoundé, Cameroon
[2]Department of Computer Science, University of Yaoundé I,
[−]P.O. Box 337 Yaoundé, Cameroon
[3]University Grenoble Alpes, CNRS, Grenoble INP ⊕, TIMA, 38000 Grenoble,
[⊕]Institute of Engineering Univ. Grenoble Alpes, France
[4]University of Yaoundé I, LIRIMA Laboratory, IDASCO Team, Cameroon
[5]Sorbonne Universities, UPMC Univ. Paris 06, IRD, UMI 209 UMMISCO, F-93143, Bondy, France

*E-mail : ndenokag@yahoo.fr

DOI : 10.46298/arima.14306
Submitted on September 19, 2024 - Published on March 25, 2025
Volume : **43** - Year : **2025**

Editors : *Mathieu Roche, Clémentin Tayou Djamegni, Nabil Gmati*

Abstract

GRAFCET is a powerful graphical modeling language for the specification of controllers in discrete event systems. It considers hierarchical structures as well as structural and semantic constraints. In this paper, we propose to use a GRAFCET specification model in a Model Driven Engineering (MDE) approach for multi-target synthesis of embedded logic control systems based on microcontrollers. In this approach, a GRAFCET metamodel is associated with a microcontroller metamodel which characterizes the microcontroller platform features to be considered when generating code. The GRAFCET metamodel includes the modeling of expressions to facilitate model verification and an easy interpretation of GRAFCET events and time constraints. Transformation rules for generation of C-programmable microcontroller code are then presented. As implementation, we present a platform based on Eclipse EMF, Object Constraint Language (OCL) and Acceleo code generation engine.

Keywords

Multi-target synthesis; logic controllers; GRAFCET; Model Driven Engineering; model verification; C code generation

I INTRODUCTION

The design cost of automated systems is greatly influenced by the time needed for the development of reliable control code [12]. This task is generally accomplished by direct implementation from the functional design specification of the controller. Conventional manual translation of the requirements of the control software into a control code often leads to additional costs

caused by erroneous interpretations [10, 12]. It is therefore of great interest to automatically generate software code on the basis of a graphical specification language [8] such as GRAFCET that is an international standard (IEC 60848 [2]) since 1988. Indeed, it is an advantageous graphical modeling language for industrial programmable logic controller (PLC) specification in discrete event systems (DES) [8].

A lot of work has been done to make GRAFCET a programming language. One of the well-known developments [7] has led to the definition of Sequential Function Chart (SFC), which is one of the five languages of the IEC 61131-3 standard dedicated to the programming of PLCs. On the other hand, some authors have been interested in code generation for controllers specified in GRAFCET. For instance, J. Machado et al. [5] presented a safe controller design methodology permitting to easily generate control code for logic controllers taking as input a GRAFCET specification model. Their proposal uses GRAFCET algebraic equations as a formal representation of GRAFCET. Today the rapid advances in electronic technologies have resulted in a variety of new and inexpensive control capabilities[15]. We are observing the emergence and swift expansion of a diverse range of low-cost processors designed for executing programs in complex embedded applications. Thus, the use of programmable controllers based on microprocessors may be preferred in low cost applications to reduce the cost of the control solution. As a consequence, it is important to consider the description of the target architecture when generating code for a system specified in GRAFCET. This allows to handle the generation of control code for a family of hardware architectures, with the possibility to choose one specific architecture as input of the generation process.

Among the existing approaches for code generation from formal models, recent advances in the field of Model Driven Engineering (MDE) produce the most promising outcomes [12]. MDE is an expanding paradigm in the software engineering domain that promotes the use of models and model transformations for the production of software artifacts (documentation, code, etc.), with the use of Domain Specific Languages (DSLs). The MDE approach was found to be appropriate for GRAFCET implementation [8, 12]. Indeed, the GRAFCET language can be seen as a DSL and can benefit from the advances of MDE to facilitate control engineers practices, by enabling the automatic transformation of GRAFCET models into control code.

Whatever be the nature of the model used to represent GRAFCET models in the code generation process, this model must support the verifications that ensure compliance with the standard [5]. A step towards a general formal definition of GRAFCET is proposed in [10] and can be used as a basis for GRAFCET model-driven development [12], including hierarchical structures [17] to enable the expantion of the existing solutions to other issues of formal methods in control system engineering. Our objective in this paper is to propose a GRAFCET metamodel representing all its basic concepts including events and time constraints. We will then show how to perform multi-target code generation, considering the specification of the target used.

The rest of the paper is organized as follows: Section II presents a background on GRAFCET specification language and model driven development and analysis of MDE work for GRAFCET implementation. GRAFCET metamodeling, verification rules and the derived properties are presented in Section III. In Section IV, we present a multi target code generation, including the microcontroller metamodel while Section V is devoted to a case study. The paper is concluded in Section VI.

II BACKGROUND

The specification of the logic controller is the first step in the development of embedded controllers for a custom application [1]. GRAFCET is one of commonly used formal techniques for logic controller specification. In this section, we present an overview of the GRAFCET language and the MDE, which is the approach through which we formalize the GRAFCET multi-target synthesis.

2.1 GRAFCET description language

GRAFCET is a graphical language for modeling automation systems defined in the IEC 60848 standard [2]. It is used for high level behavioral description of logic sequential systems and has been inspired from the Petri Net language [1]. GRAFCET is used for the specification, modelling and simulation of logic control systems in interaction with physical processes. A GRAFCET model describes the states of a system and associated actions that permit to take into account inputs and generate the corresponding outputs. This language is defined statically by its syntax and dynamically by its evolution rules.

2.1.1 GRAFCET statics

A GRAFCET model (as presented in Figure 1) is a directed graph with two types of nodes: steps and transitions. Steps are represented by squares while transitions are represented by horizontal lines.

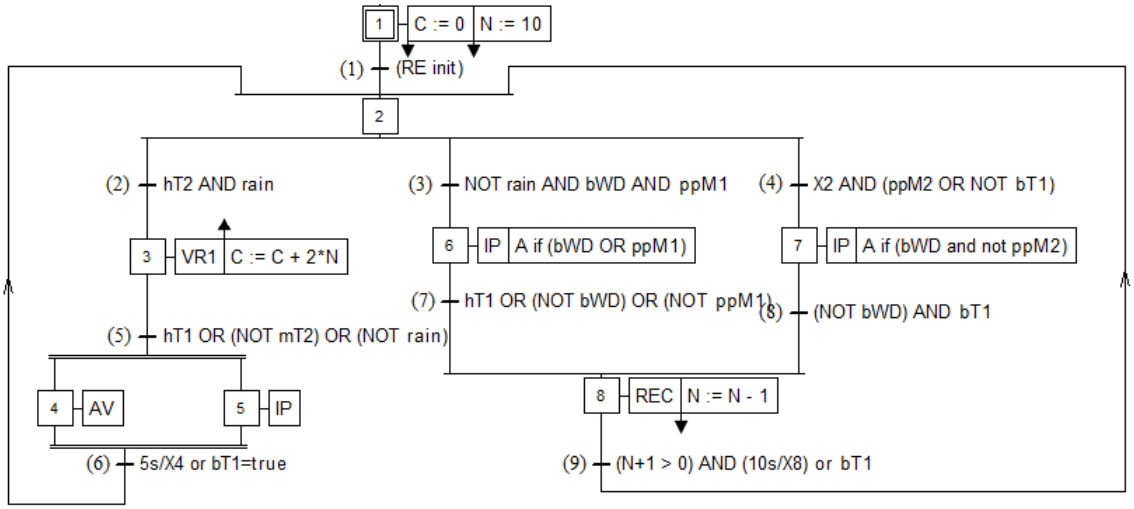

Figure 1: Example of GRAFCET model

Initial steps are represented with double squares. Steps can be either numbered or named, while transitions do not require numbering. Steps and transitions are linked by directed arcs, referred to as junctions or connections. These arcs are essential for connecting steps to transitions and vice versa. Each transition is associated with a transition condition, also known as receptivity or condition.

2.1.2 GRAFCET dynamic behavior

The GRAFCET dynamic behavior can be compared to a sequential machine that provides an event-driven conversion of an input sequence into a set of outputs, considering the controller's internal state [1, 5]. The GRAFCET evolution is possible by firing (or clearing) transitions

according to five evolution rules defined by the IEC 60848 standard [2] which aims to ensure a deterministic behavior :

- **Rule 1**: At the initial time, all the initial steps are active; all the other steps are inactive.
- **Rule 2**: A transition is enabled when all the steps that immediately precede this transition are active. A transition is fireable when it is enabled and when the associated transition condition is true. A fireable transition must be immediately fired.
- **Rule 3**: Firing a transition provokes simultaneously the activation of all the immediately succeeding steps and the deactivation of all the immediately preceding steps.
- **Rule 4**: When several transitions are simultaneously fireable, they are simultaneously fired.
- **Rule 5**: When a step shall be both activated and deactivated, by applying the previous evolution rules, it is activated if it was inactive, or remains active if it was previously active.

These rules enable the calculation of the subsequent state and the corresponding output signals caused by an input event [1, 2, 12]. A step defines a partial state of the system and can be active or inactive; hence, a Boolean variable X_i, named step activity variable is defined for each step. The variable X_i (which is an internal variable) is true (1) if the step i is active and false (0) if not. The general state of a GRAFCET model called its situation, is characterized by the set of all the active steps at a given time. It can be represented by a vector $X = (X_i)$. Initial steps represented by double squares are initially activated (*Rule 1*). As soon as time passes and events occur, the continuous changing of the GRAFCET situation characterizes the evolution of the system that it models. A mathematical formalisation of these dynamics is proposed in [10] and R. Mross et al. [19].

2.1.3 GRAFCET example

Figure 1 shows an example of a GRAFCET model used in [18], inspired from a model presented in [13] to model the water supply subsystem of a tank. This model has eight steps numbered from 1 to 8 among which the step 1 is initial, nine transitions numbered from (1) to (9) and several actions among which : $C := 0$ and $N := 10$ are stored actions performed during the deactivation of step 1; $VR1$, AV and REC are continuous level actions associated respectively to steps 3, 4 and 8. The action *A if (bWD OR ppM1)* is a conditional level action. It is performed if step 6 is active and the condition *bWD OR ppM1* is true. The receptivity of transition (2) is *hT2 AND rain*. It expresses the fact that when step 2 is activated and the value of *hT2 AND rain* is $true$, this transition is fireable and should be fired; when it is fired, step 2 is deactivated and step 3 is activated. Here, $hT2$ and $rain$ are two Boolean variables representing digital input signals.

2.2 Model driven engineering

Model Driven Engineering (MDE) is the field of software engineering that makes use of models and model transformation to produce software artifacts such as code and documentation [11].

2.2.1 Key principles and MDE approaches

The basic principle of MDE is *"everything is a model"* [9, 11]. A model is a representation of a system under study. MDE principles state that a particular view of a system can be captured by a model and each model is written in the language of its metamodel. In other words, "a metamodel is a model of models" that defines the structure of a modeling language [11]. As a consequence, a model should satisfy the structure defined at the level of its metamodel. A modeling language

is a set of all possible models that are conforming to the modeling language's abstract syntax, represented by one or more concrete syntaxes and satisfying a given semantics [11]. The process of defining a modeling language starts with the identification of the concepts, abstractions and relations underlying the application domain. It corresponds to the domain analysis phase of the development of a Domain Specific (Modeling) Language (DS(M)L).

MDE approaches are usually supported by complex tools called "model driven MetaTools" and commonly known as "language workbenches" [16]. They provide a collection of features to help users define DS(M)Ls, with specific editors, model validation and model transformation. Examples of such tools are Eclipse Modeling Framework (EMF), Microsoft Software Factories, and JetBrains MPS [11]. A model transformation is "the process of converting one model to another model of the same system'" [11]. A model transformation program takes as input a model conforming to a given source metamodel and produces as output another model conforming to a target metamodel [16].

2.2.2 Related work

Many PLC environments such as CoDeSys allow multi-target synthesis of logic control systems [10], but these environments are proprietary and they are not interested by the synthesis on microcontroller targets. Y. Qamsane et al [14] proposed a GRAFCET metamodel for the transformation of Distributed Control model of automated manufacturing systems into GRAFCET to facilitate its implementation. This model represents the very basic GRAFCET structure, but is limited to allow the construction of any GRAFCET model. For example, only one action can be associated to a step, and its type (continuous or stored) is not taken into consideration. Similarly, to demonstrate that composing transformations is a complex problem, F. Basciani et al. [9] proposed a GRAFCET metamodel to illustrate model transformations between incompatible metamodels, with an illustration on the transformations between GRAFCET and Petri nets. This GRAFCET metamodel conforms to the GRAFCET standard and represents only the concepts of the most basic structure of the language. Similarly, R. Julius et al. [17] proposed a metamodel based approach for GRAFCET specifications, with a particular focus on hierarchical structures, enabling how to expand the existing solutions to other issues of formal methods in control system engineering. The variable and timing condition concept is presented, discussed and formalized by G. Nzebop N. et al. [18]. Their proposal integrates a parser capable of directly analysing and generating GRAFCET expressions in an MDE environment for editing GRAFCET models. However, this solution had not yet been integrated into a general GRAFCET metamodel. Recently, R. Mros et al. [19] proposed a GRAFCET metamodel for editing models and transforming them into Guarded Action Language (GAL) for verification purposes. Their contribution emphasizes the hierarchical structures of GRAFCET and rules for editing valid GRAFCET models. Also, the GRAFCET expressions must be transformed into GAL before being verified and validated. Another limitation of this MDE synthesis solution is that their target is mainly programmable controllers (via the Structured Text, one of the PLC languages [7]) and does not take into account specific targets such as microcontrollers.

Here we propose a metamodel that allows the editing of valid GRAFCET models with well-constructed and verified expressions [18, 19] based on a GRAFCET expression parser and the OCL language, associated with a metamodel of C-programmed microcontrollers [4] to facilitate code synthesis for these architectures.

III GRAFCET CONCEPTS AND METAMODEL

Here, we present the GRAFCET metamodeling, consisting of the identification of GRAFCET concepts with their interrelations, and their formalization within a metamodel.

3.1 GRAFCET concepts identification

Given the complexity of the GRAFCET domain, we distinguish the identification of concepts of the basic GRAFCET structure, concepts related to variables and actions, GRAFCET expressions concepts and timing variables concepts.

3.1.1 Concepts of the basic GRAFCET structure

With regard to the description of the GRAFCET language according to the IEC 60848 3rd Ed. standard [2], it appears that a GRAFCET model groups together several steps and transitions. They are linked together by oriented links also called connections. The steps (Step concept), the transitions (Transition), oriented links (*Connection*) and variables (*Variable*) are GRAFCET elements (*G7Element*). Two types of oriented links can easily be identified: transition-to-step link (*TransitionToStep*) and step-to-transition link (*StepToTransition*). Each instance of *TransitionToStep* is outgoing from a transition and incoming from a step, while each instance of *StepToTransition* is outgoing a step and incoming a transition.

3.1.2 Concepts related to variables, actions and expressions

A Boolean variable (*BooleanVariable*) is associated with a step to represent its activity, and is internal to the GRAFCET. Any variable (*Variable*) is either input, output or internal. It is characterized by a name and a duration of its activity. Several actions (*Action*) may be associated with a step. Every action is represented and performed by its variable. This way of structuring Action and Variable concepts makes it possible to have the same action associated with several different steps as stated in the standard. An action can only be stored (*StoredAction*) or level (*LevelAction*).

The concept *Expression* do not appear explicitly in the GRAFCET standard, but it exists and its modeling permits to solve certain issues such as verifications and the providing of appropriate semantics. This concept is presented, discussed and formalized by G. Nzebop N. et al. [18], including GRAFCET events and timing variables. In [19], the authors also identify the notion of *Variable* as a key concept, and use the concept *Condition* to refer to Boolean expressions (*Expression*).

3.1.3 Concepts related to timing variables

Here is an overview of the concept of timing variables, as detailed in [18], and extended to other Boolean variables within GRAFCET, including suitable semantics proposed in the C programming language. The GRAFCET standard defines timing variables or conditions, exemplified by *X1/3s* (termed Delayed 1), *not X1/3s* (referred to as Limited), and *2s/X1/3s* (known as Delayed 2, serving as the general form), all derived from the activity variable X1. These expressions utilize a variable, and their interpretation involves calculations. For instance, the condition *X1/3s* holds true if the duration since variable $X1$ became true is at least 3 seconds. The determination of timing variable values relies on tracking the duration of the associated variable's activity.

Thus, each GRAFCET variable must be defined by its state (active or inactive) and its lifetime, which is the duration since it transitioned from false to true. We employ the concept of *TimingOperator* to represent a timing variable, which includes a type (*Limited*, *Delayed1*, or

Delayed2) and two durations (*duration1* with unit *unit1* and *duration2* with unit *unit2*); *duration2* is applicable only for *Delayed2* timing expressions. Units of duration are specified in the enumerated type *TimeUnit*.

3.2 The GRAFCET metamodel

The formalization of GRAFCET concepts and the links between them produces the GRAFCET metamodel of Figure 2, where the elements depicted in purple represent components of the main GRAFCET structure. The yellow elements illustrate aspects of the GRAFCET expression language '[18], while the blue elements refer to enumerated types or potential values for specific attributes within the model.

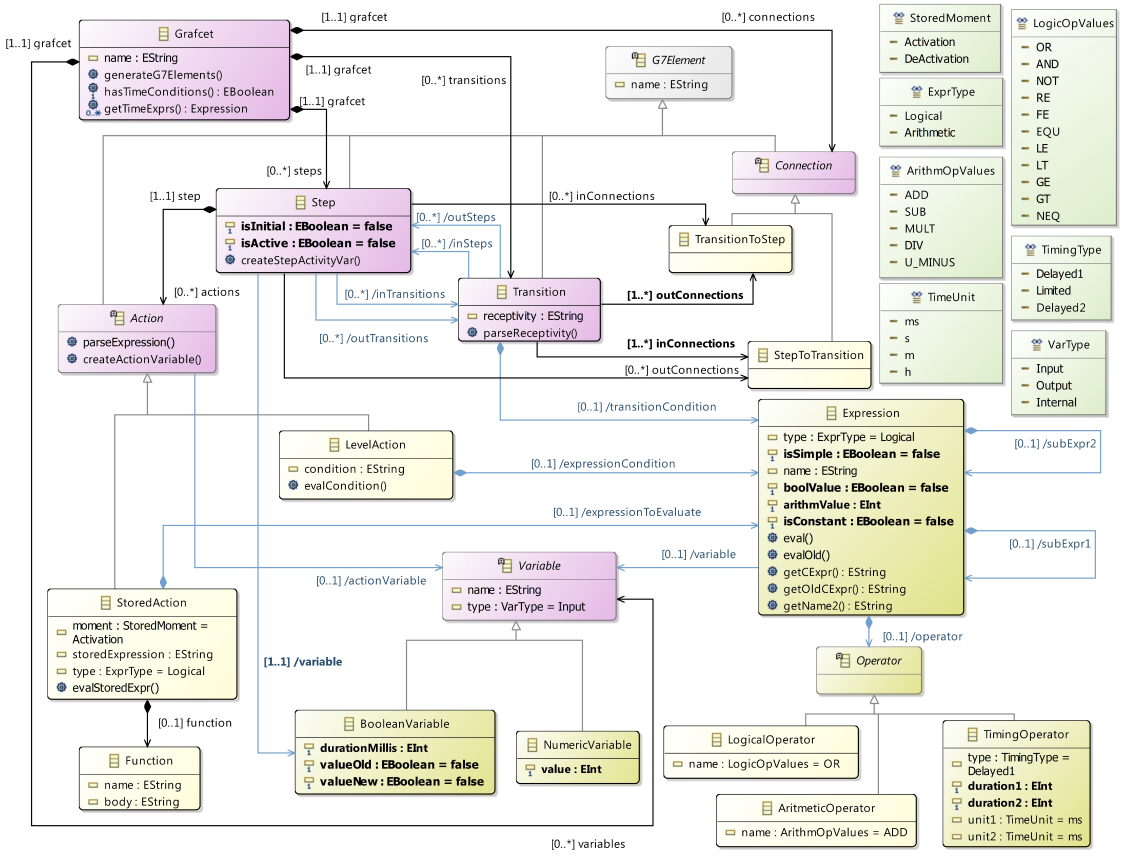

Figure 2: The GRAFCET metamodel

Many relationships are automatically derived. An automatic solution for the construction of GRAFCET expressions has been presented in [18], based on an ANTLR parser generator tool, called *G7Expr*, that is called to automatically and recursively derive all the elements associated with the construction of GRAFCET expressions. We therefore integrate this solution into the proposed GRAFCET metamodel.

To ensure that the built models are valid, semantic constraints or rules are stated, formalized with OCL (in the same way as [19] and [18]) and integrated into the GRAFCET metamodel. In effect, OCL is a formal language that is independent of a programming language [19]. It is used to describe elements on UML models and to query metamodel instances. Generally, OCL expressions are written in the context of a specific instance of a model, to which the keyword

self refers. The "." operator refers to an attribute, resulting to a single attribute or a set, called collection; while the "->" operator refers to the navigation from a collection.

For example, the constraint "*A GRAFCET has at least one initial step*" is formalized with OCL as follows :

Listing 1: A GRAFCET has at least one initial step

```
context Grafcet invariant hasAtLeastOneInitialStep :
      self.steps->select(s|s.isInitial)->size()>=1;
```

Annex 1 contains other rules that have been clearly identified, stated and formalized with OCL.

3.3 Deriving relative positions between steps and transitions

The GRAFCET evolution rules (defined in Section 2.1.2) make use of relative positions between steps and transitions. For example, given a transition, it is necessary to evaluate all the input steps (upstream steps) and all the output steps (downstream steps). We provide a solution by using the OCL language to query metamodel instances.

Input steps of a transition : According to the model, a step is at the input of a transition if there exists a link (of type StepToTransition) which is both at the output of this step and at the input of this transition. Input steps are obtained by creating the *inSteps* property in the context of Transition as presented on Listing 2:

Listing 2: Deriving inSteps property

```
property inSteps:Step[*] { derived volatile }
{ derivation: (grafcet.steps->select(step|step.outConnections->exists(
    outCon|self.inConnections->includes(outCon))))->asSet(); }
```

Similarly, we create derived properties for output steps of a transition, input transitions of steps, and output transitions of steps, all of which are required to implement the GRAFCET evolution rules.

IV MULTI TARGET CODE GENERATION

This section starts with the specification of target platforms, before describing the transformation of GRAFCET into control code.

4.1 Target platforms specification and metamodel

The microcontrollers programmable in a language derived from the C-language [4] are considered, and the metamodel of this family is drawn in Figure 3. As for the GRAFCET, a microcontroller model editor is also obtained in Eclipse EMF, allowing the edition and saving models in XMI format for any use, such as code generation.

The elements shown in white represent useful physical characteristics, while those in purple illustrate characteristics associated with the C-language. The light yellow elements denote enumerated types or potential values for specific attributes within the model.

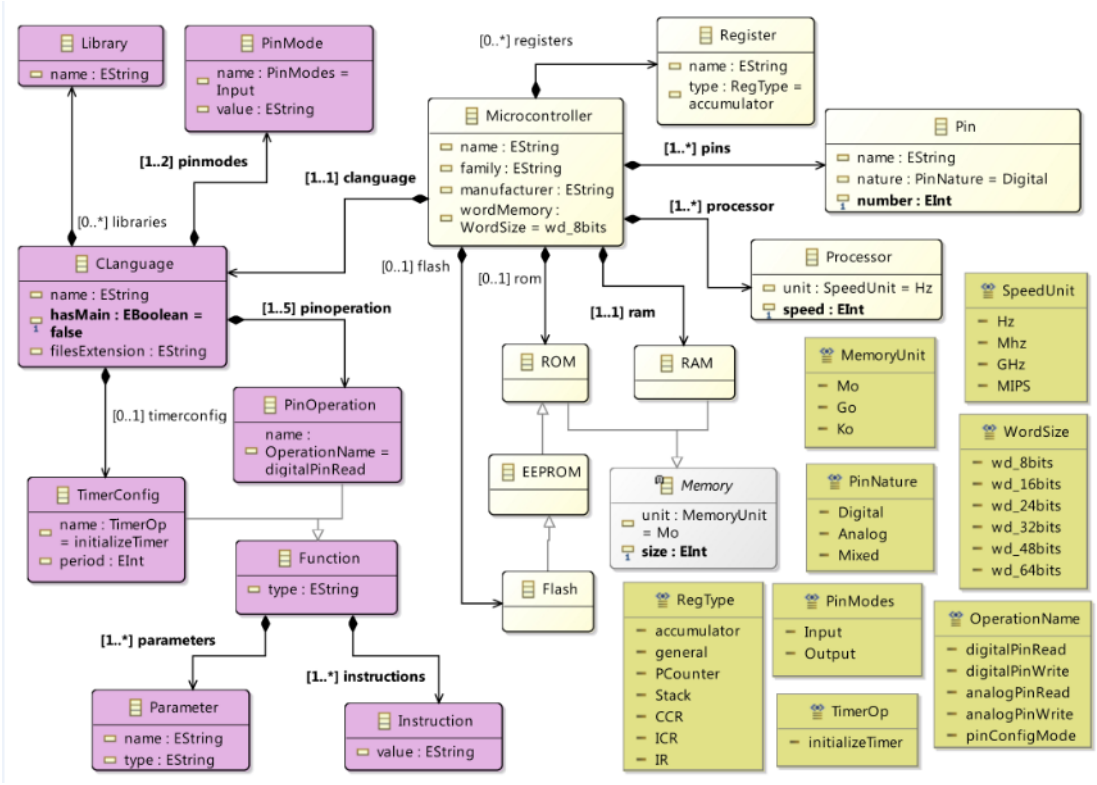

Figure 3: Microcontroller metamodel

4.2 Transformation step for code generation

Here is the MDE transformation process to generate GRAFCET code, which utilizes both the GRAFCET model and the target microcontroller model as input. We then present a formal tool for describing the semantics of GRAFCET, specifically the algebraic equations, followed by the general structure of the C code to be generated for controlling the system modeled by GRAFCET, along with the associated transformation rules.

4.2.1 GRAFCET algebraic equations

Due to the sequential execution of instructions by microcontrollers, the GRAFCET dynamics is stated in the code in terms of GRAFCET algebraic equations presented in [5] and recalled in [13]. Here are the two main equations of the GRAFCET dynamics. Let $CC(tr)$ (Clearing Condition) be the Boolean variable associated to the clearing of transition tr: tr can be fired if it is validated and if its associated transition condition $TC(tr)$ is true. $CC(tr)$ is calculated as shown on equation 1.

$$CC(tr) = (\prod_{i=1}^{m} X_i^{tr}) \times TC(tr) \tag{1}$$

where :

- X_i^{tr} is the step activity Boolean variable associated to step i and directly preceding transition tr,
- $TC(tr)$ is the transition condition associated to transition tr and
- m is the number of steps immediately preceding the transition tr.

$\prod_{i=1}^{m} X_i^{tr}$ expresses the condition for this transition to be validated.

After the initialization of activity variables, their update is computed as shown on equations 2.

$$X_i(t+1) = \sum_{j=1}^{p} CC(tr_j^{i-}) + X_i(t) \times \prod_{j=1}^{q} \overline{CC(tr_j^{i+})} \tag{2}$$

where :

- $X_i(t)$ is the step activity variable of step i in the t^{th} scan cycle,
- $X_i(t+1)$ is the step activity variable of step i in the $(t+1)^{th}$ scan cycle,
- p is the number of transitions directly preceding the step i,
- q is the number of transitions directly succeeding the step i,
- $CC(tr_j^{i-})$ is the clearing condition of transition j, directly preceding the step i and
- $CC(tr_j^{i+})$ is the clearing condition of transition j, directly succeeding the step i.

To calculate the value of the actions, a Boolean variable A is associated with each action $\mathcal{A}$. Since it's possible for the same action $\mathcal{A}$ to be associated with several steps, the value of $A(t)$ is obtained by calculating the logical $OR(+)$ of the step variables $X_i^{\mathcal{A}}, i = 1, 2, \ldots, h$; where h is the number of steps with which this action is associated (equation 3):

$$A(t) = \sum_{i=1}^{h} X_i^A(t) \tag{3}$$

Where $X_i^A(t)$ is the activity variable of a step i at time t to which the action A is associated.

The equation 3 proposed in [5] concerns only continuous level actions (with 1 or *true* condition). This equation is generalized for conditional level actions, by combining each step activity variable with which the action is associated with the corresponding condition. The result is equation 4 :

$$A(t) = \sum_{i=1}^{h} (X_i^A(t) \times Cond_{X_i}^A(t)) \tag{4}$$

where $Cond_{X_i}^A(t)$ is the condition at time t of the A action associated with the step whose activity variable is X_i.

For stored actions, these are textual expressions primarily intended for variable assignment, either when the step is activated or when it is deactivated. Concerning GRAFCET expressions, they are transformed into C code using the C semantics of expressions described in [18] and produced by a GRAFCET expression parser.

4.2.2 *General structure of the generated code*

The transformation is based on the correspondence of GRAFCET-elements to C code fragments (M2T transformation) of the *Concrete syntax* design pattern category. For M2M transformations, it is possible to thoroughly describe how each element of the source model is transformed into the target model, which is not always straightforward for M2T transformations. The overall structure of the generated code is presented in Listing 3 :

Listing 3: General structure of the generated code

```
// Inclusion of necessary libraries for the specific target
// Declaration of variables for the Grafcet model
void setup() {
   initializeTimer(); // Initialize timer if the Grafcet uses timing
       conditions
   // Configure input and output pins
   // Set the initial state of the Grafcet
   X1 = 1; // Set X1 as the initial step of the Grafcet model
}
void loop() {
   // Read the input signals
   // Evaluate validated transitions (VT_i)
   // Assess receptivities (R_i)
   // Calculate clearing transition conditions (CC_i)
   // Determine the new state of the Grafcet (Xi)
   // Calculate outputs (level actions)
   // Evaluate stored actions
   // Updating outputs
}
// Include this if the C language compiler requires a main function
void main(void) {
   setup();
   while (1) { loop(); }
}
void initializeTimer(){...}
void update_G7TimingVars_callback(){...}
void pinModeConfig(int pin_num, int mode){ ... }
... //other functions of reading/writing pins
```

For additional details, *Annex 2* offers a summary of several essential code transformation rules. Specifically, Equation 1 is realized through the rules in Listing 13 and Listing 14, while Equation 2 is realized by the rule in Listing 15.

V A CASE STUDY OF CODE GENERATION

This case study aims to provide an example of implementing transformations for code generation specifically for the family of microcontrollers discussed here. We then present a particular case with the *Atmega328P* microcontroller [21].

5.1 An implementation of the transformation with Acceleo

The transformation program is organized by modules. Each module contains several templates and/or queries to extract information from the manipulated models and write the result into the file on output. The Acceleo language is then used to implement the transformation of

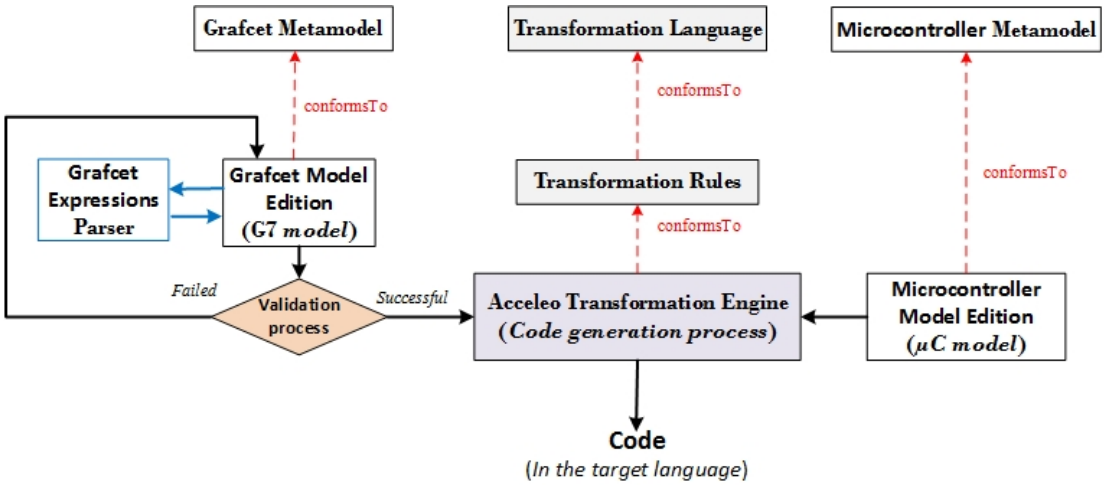

Figure 4: General architecture of the transformation system

GRAFCET into code. It is an implementation of the MOFM2T specification defined by the OMG and is made up of two main types of structures: templates and queries. Templates are sets of Acceleo statements that are used to generate text, and queries are used to extract information from models.

The main module (*generateG7MM2Code.mtl*) contains one template that provides the main structure of the code generated and outputted in a file, as shown in Figure 8 of *Annex 3*.

The general architecture of this transformation system is shown in Figure 4. The *Acceleo Transformation Engine* takes as input a valid GRAFCET model and a description of the architecture of the target microcontroller to execute the transformation rules and produce dedicated code as output.

5.2 Editing and validation of the GRAFCET model

After the creation of the generation model (*.genmodel*) within Eclipse EMF, the project code is automatically generated, including the code of a GRAFCET editor, which has several views including a tree editor (*Sample Reflective Ecore Model Editor*) and a text editor. This is a significant advantage of using a MDE environment, as it provides readily available model manipulation tools.

Here we illustrate this editor with the GRAFCET model from the example (Figure 1). Based on this GRAFCET model, we present a corresponding GRAFCET model with labeled links (as shown in Figure 5), enabling a clear differentiation between the links.

The labels (*con1*, *con2*, etc.) permit to distinguish links from each other. This GRAFCET model has a total of 20 links: 10 of type *StepToTransition* and 10 of type *TransitionToStep*. All the derived features of this GRAFCET model are automatically produced, according to Section 3.3. Figure 6 shows an overview of this GRAFCET model produced in the *Sample Reflective Ecore Model* editor. All step activity variables ($X1$, $X2$, ..., $X8$) are automatically built, as well as step variables, actions, and transition expressions.

Validation rules are associated with the GRAFCET metamodel and must be verified in model instances by running the validation process.

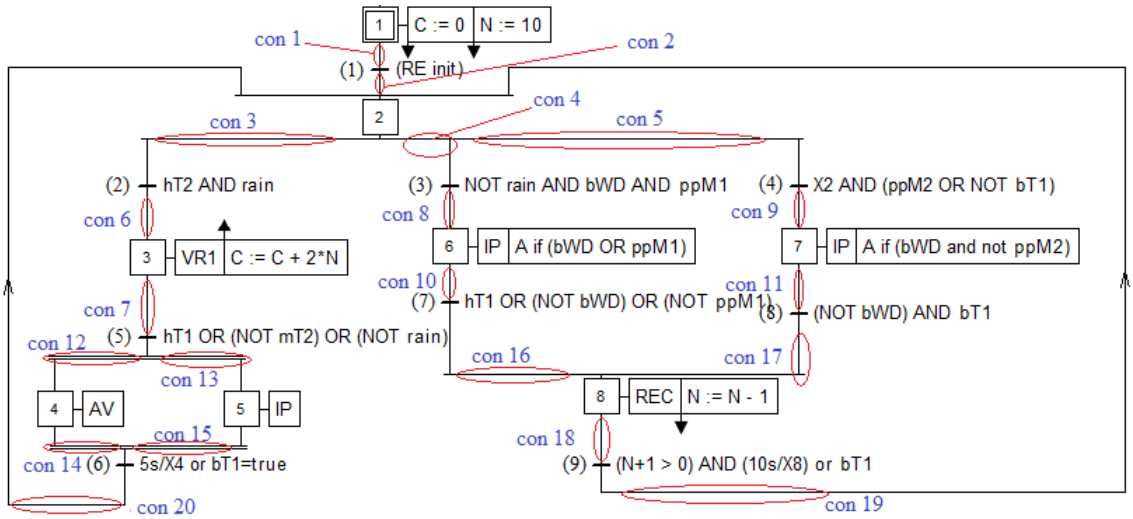

Figure 5: GRAFCET example with the links labelled

5.3 Application to the *Atmega328P* microcontroller

5.3.1 Atmega328P microcontroller description and attributes

Atmega328P is a high performance Microchip 8-bit microcontroller based on the AVR enhanced RISC architecture, manufactured by the Atmel company [21]. The Atmega328P is a product of open-source hardware projects, driven by the development of platforms known as Arduino [6, 15]. This microcontroller powers the Arduino Uno development platform. As an open-source hardware project, all the specifications of the circuit board and electronic components, along with the IDE software, are freely accessible for anyone to use or modify [15]. The attributes of the *Atmega328P* used for code generation are listed as follows:

- Name: *Atmega328P*, Manufacturer: ATMEL, 8 bits word memory;
- 20MHz of processor, 2Ko of RAM, 32Ko of Flash memory, 1Ko of EEPROM;
- Programmable pins with numbers: PD0 (0) ...PD7(7), PB0 (8) ...PB7 (15), PC0 (0) ...PC5(28);
- C-language characteristics: Name: Arduino, Timer: Timer 1 of 16 bits;
- Pins operations :
  `pinMode(pin_num, mode);` to configure a pin number with a particular mode (INPUT/OUTPUT),
  `digitalRead(pin_num);` to read a digital value of a pin number,
  `digitalWrite(pin_num, value);` to write a digital value on a pin number,
  `analogRead(pin_num);` to read an analog value of a pin number,
  `analogWrite(pin_num, value);` to write an analog value on a pin number;
- Timer 1 configuration: `Timer1.initialize(1000000/(1000/TIMER_PERIOD));`
  `Timer1.attachInterrupt(update_G7TimingVars_callback);`
  To configure the Timer 1 (16 bits timer) with a period of *TIMER_PERIOD* milliseconds. It calls periodically the function `update_G7TimingVars_callback`.

The ecore metamodel instance corresponding to ATmega328P is produced and used in the code generation process. An overview of the metamodel instance corresponding to ATmega328P is given in Figure 7. The left part represents the physical characteristics, displaying, for example, the 23 pins numbered from PD0 to PC5. The right part illustrates how the Arduino language interacts with these pins and configures a timer.

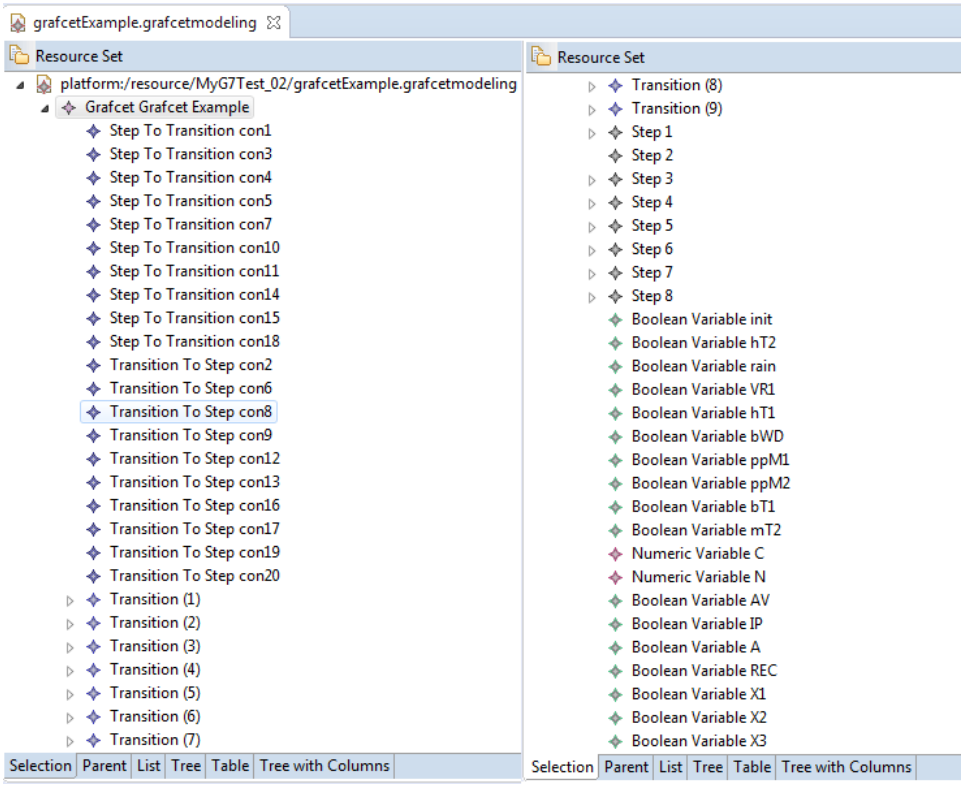

Figure 6: The GRAFCET example in tree editor

5.3.2 Generation of GRAFCET code in Arduino language

An implementation of the *M2T transformation* has been executed to generate arduino code. After the selection of the GRAFCET model, the microcontroller instance and the target directory, the transformation program is run and the target code is produced. An overview of the resulting code is presented in *Annex 4*.

This generated code compiles successfully in the Arduino environment and runs on any Arduino board (such as Uno, Mega, ...) equipped with the Atmega328P microcontroller, producing the expected behavior.

VI CONCLUSION AND REFERENCES

6.1 Discussion

In this paper, we first introduce a GRAFCET metamodel along with associated rules that facilitate the creation of valid GRAFCET models characterized by well-structured and verified expressions [18, 19]. This is achieved through the integration of a GRAFCET expression parser and the Object Constraint Language (OCL). Next, we introduce a metamodel for C-programmable microcontrollers [4], along with transformation rules aimed at optimizing code synthesis for these controller architectures.

All verifications of the input GRAFCET model are conducted within the synthesis environment. This is achieved using the expression parser, which guarantees the accurate construction of expressions —including complex timing expressions [18]— by recursively generating the corresponding instances of Expression.

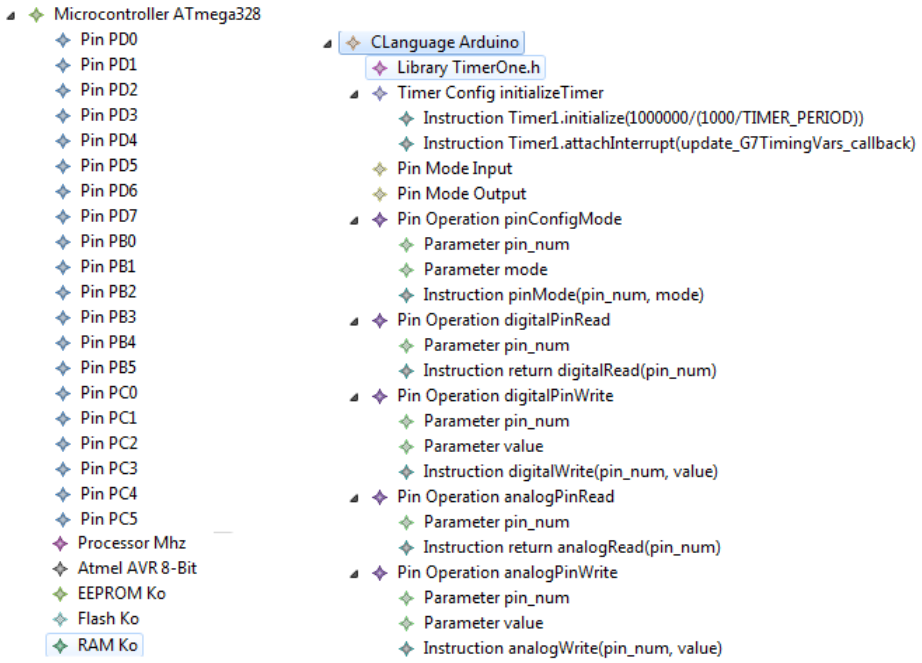

Figure 7: Microcontroller model instance (Atmega328P)

Additionally, the rules defined and formalized in OCL are executed by the GRAFCET editor, which is generated by the MDE environment in Eclipse EMF [16].

While timing conditions are thoroughly addressed, the GRAFCET structures (macro steps, enclosing steps, and forcing orders [2]) are not defined as in [19], where the metamodel is only partially presented. However, this limitation is mitigated by the fact that any GRAFCET model featuring hierarchical structures can be transformed into an equivalent flat GRAFCET model, often referred to as a sound GRAFCET [3, 19].

The extension of MDE-based controller synthesis to microcontroller targets allows us to accommodate a broad range of hardware architectures beyond traditional PLCs. This is particularly relevant as, for certain applications, programmable controllers based on microprocessors may be preferred over PLCs to reduce overall control solution costs. By combining the functional specification with the description of the microcontroller target, we can develop integrated MDE platforms that enable the safe synthesis of low-cost controllers. The code generated for one target can be readily adapted for another C-programmable target. If needed, the transformation itself can be reused with minimal modifications to accommodate the new target, ensuring multi-target synthesis. Furthermore, the transformation rules outlined in this paper can be easily adapted to generate code in programming languages other than C, thus supporting multi-languages code generation, using an approach such as that presented in [20].

6.2 Conclusion

The objective of this paper was to explore the multi-target synthesis of logic embedded controllers from GRAFCET specifications. We have proposed a GRAFCET metamodel that considers all the basic concepts of the GRAFCET language, including time constraints and events. This has led to the creation of a GRAFCET metamodel associated with a GRAFCET expression parser, facilitating the design of verified GRAFCET models. To allow multi-target generation, we have proposed a microcontroller metamodel representing its main characteristics useful for

code generation. Transformation rules have been designed for GRAFCET code generation, given the model of the target microcontroller, with an implementation case study in the popular Eclipse MDE environment. The flexibility of the multi-target platform for embedded control synthesis, proposed in this paper, allows PLC technology to be used in a wide variety of applications that were not previously associated with PLCs. The proposal presented in this paper is fully transparent and can be easily adapted for any other purpose.

Exploring microcontrollers with simplified non-C programming appears worthwhile. Future research could extend the target metamodel and transformation rules to enable multi-target and multi-language synthesis.

REFERENCES

Publications

[1] R. David. "Grafcet: A powerful tool for specification of logic controllers". In: *IEEE Transactions on control systems technology* 3.3 (1995), pages 253–268.

[2] I. E. Commission. *IEC 60848: GRAFCET specification language for sequential function charts*. Technical report. Tech. rep. International Electrotechnical Commission, 2002.

[3] R. David and H. Alla. *Discrete, continuous, and hybrid Petri nets*. Volume 1. Springer, 2005.

[4] O. Bayó-Puxan, J. Rafecas-Sabaté, O. Gomis-Bellmunt, and J. Bergas-Jané. "A GRAFCET-compiler methodology for C-programmed microcontrollers". In: *Assembly Automation* 28.1 (2008), pages 55–60.

[5] J. Machado, E. Seabra, J. C. Campos, F. Soares, and C. P. Leão. "Safe controllers design for industrial automation systems". In: *Computers & Industrial Engineering* 60.4 (2011), pages 635–653.

[6] F. Daniel K and G. Peter J. "Open-source hardware is a low-cost alternative for scientific instrumentation and research". In: *Modern instrumentation* 2012 (2012).

[7] IEC61131-3. "Programmable controllers—Part 3: programming languages (3rd ed.)" In: *International Electrotechnical Commission publishing* (2013).

[8] F. Schumacher, S. Schröck, and A. Fay. "Tool support for an automatic transformation of GRAFCET specifications into IEC 61131-3 control code". In: *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE. 2013, pages 1–4.

[9] F. Basciani, D. Di Ruscio, L. Iovino, and A. Pierantonio. "Automated chaining of model transformations with incompatible metamodels". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2014, pages 602–618.

[10] F. Schumacher and A. Fay. "Formal representation of GRAFCET to automatically generate control code". In: *Control Engineering Practice* 33 (2014), pages 84–93.

[11] A. R. Da Silva. "Model-driven engineering: A survey supported by the unified conceptual model". In: *Computer Languages, Systems & Structures* 43 (2015), pages 139–155.

[12] R. Julius, M. Schürenberg, F. Schumacher, and A. Fay. "Transformation of GRAFCET to PLC code including hierarchical structures". In: *Control Engineering Practice* 64 (2017), pages 173–194.

[13] G. N. Ndenoka, E. Simeu, and R. Alhakim. "Efficient controller synthesis of multi-energy systems for autonomous domestic water supply". In: *Revue Africaine de Recherche en Informatique et Mathématiques Appliquées* 24 (2017).

[14] Y. Qamsane, M. El Hamlaoui, A. Tajer, and A. Philippot. “A Model-Based Transformation Method to Design PLC-Based Control of Discrete Automated Manufacturing Systems”. In: *Proceedings of Engineering and Technology–PET* 19 (2017), pages 4–11.
[15] P. Zabala, M. C. Abas, and P. Cerna. “‘Development of programmable relay switch using microcontroller”. In: *American Journal of Remote Sensing* 5.5 (2017), pages 529–551.
[16] M. Soukaina, B. Abdessamad, and M. Abdelaziz. “Model Driven Engineering (MDE) Tools: A Survey”. In: *American Journal of Science, Engineering and Technology* 3.2 (2018), page 29.
[17] R. Julius, T. Trenner, A. Fay, J. Neidig, and X. L. Hoang. “A meta-model based environment for GRAFCET specifications”. In: *2019 IEEE International Systems Conference (SysCon)*. IEEE. 2019, pages 1–7.
[18] G. N. Ndenoka, M. Tchuenté, and E. Simeu. “Langage et sémantique des expressions pour la synthèse de modèle Grafcet dans un environnement IDM”. In: *Revue Africaine de Recherche en Informatique et Mathématiques Appliquées* 33 (2021).
[19] R. Mross, A. Schnakenbeck, M. Völker, A. Fay, and S. Kowalewski. “Transformation of GRAFCET into GAL for verification purposes based on a detailed meta-model”. In: *IEEE Access* 10 (2022), pages 125652–125665.
[20] T. Xue, X. Li, T. Azim, R. Smirnov, J. Yu, A. Sadrieh, and B. Pahlavan. “Multi-Programming Language Ensemble for Code Generation in Large Language Model”. In: *arXiv preprint arXiv:2409.04114* (2024).
[21] Atmel. *ATMega328 datasheets*. Accessed Apr. 2024.

ANNEX 1 : SEMANTIC CONSTRAINTS OF GRAFCET

uniqueNamesInVars constraint: Two different variables cannot have the same name :

Listing 4: uniqueNamesInVars constraint (Grafcet)

```
context Grafcet invariant uniqueNamesInVars:
   self.variables->forAll(v1,v2| v1<>v2 implies v1.name<>v2.name);
```

validTransition constraint: Any transition has at least one step in input and one step in output :

Listing 5: validTransition constraint (Transition)

```
context Transition invariant validTransition :
      self.inConnections->size()>=1 and self.outConnections->size()>=1;
```

stepVarIsInternalVar constraint: Any variable associated to a step (step activity variable) is an internal variable :

Listing 6: stepVarIsInternalVar constraint (Step)

```
context Step invariant stepVarIsInternalVar:
      self.stepVariable.type = VarType::Internal;
```

LevelActionVarIsBoolVar constraint: Any variable representing a level action is of type *BooleanVariable* :

Listing 7: levelActionVarIsBoolVar constraint (LevelAction)

```
context LevelAction invariant levelActionVarIsBoolVar:
      self.actionVariable.oclIsTypeOf(BooleanVariable);
```

validStepToTransitionStepSide constraint: An instance of *StepToTransition* can only link one step to one transition, i.e. only one incoming step :

Listing 8: validStepToTransition_StepSide constraint (Grafcet)

```
context Grafcet invariant validStepToTransition_StepSide :
   self.connections->select(c|c.oclIsTypeOf(StepToTransition))
   ->forAll(con|self.steps->select(s|s.outConnections
   ->includes(con))->size()=1);
```

validStepToTransitionTransitionSide constraint: An instance of *StepToTransition* can only link one step one transition, i.e. only one outgoing Transition :

Listing 9: validStepToTransition_TransitionSide constraint (Grafcet)

```
context Grafcet invariant validStepToTransition_TransitionSide :
   self.connections->select(c|c.oclIsTypeOf(StepToTransition))
   ->forAll(con|self.transitions->select(t|t.inConnections->includes(con))
      ->size()=1);
```

validTransitionToStepTransitionSide constraint: An instance of *TransitionToStep* can only link one transition to one step, i.e. only one outgoing Step :

Listing 10: validTransitionToStep_TransitionSide constraint (Grafcet)

```
context Grafcet invariant validTransitionToStep_TransitionSide :
   self.connections->select(c|c.oclIsTypeOf(TransitionToStep))
   ->forAll(con|self.transitions->select(t|t.outConnections
   ->includes(con))->size()=1);
```

validTransitionToStepStepSide constraint: An instance of *TransitionToStep* can only link one transition to one step, i.e. only one incoming Transition :

Listing 11: validTransitionToStep_StepSide constraint (Grafcet)

```
context Grafcet invariant validTransitionToStep_StepSide :
   self.connections->select(c|c.oclIsTypeOf(TransitionToStep))
   ->forAll(con|self.steps->select(s|s.inConnections
   ->includes(con))->size()=1);
```

ANNEX 2 : SOME BASIC TRANSFORMATION RULES

The rules are outlined in the following listings:

Listing 12: Receptivity calculation

```
R_[aTransition.name/] = <aTransition.getCExpr()>;
```

Listing 13: Validate transition (VT) computation (Acceleo)

```
//Evaluate validated transitions (variables)
[for (trans : Transition | g7.transitions)]
      VT_[trans.name/] = [for (step : Step | trans.inSteps) separator (' &&
          ') after (';')] [step.variable.name/] [/for]
[/for]
```

Listing 14: Clearing a transition (CC) computation

```
//for every transition <trans>
CC_[trans.name/] = VT_<trans.name> && R_<trans.name>;
```

Listing 15: Steps activity variables(Xi) computation

```
[for (step : Step | g7.steps)]
      [step.variable.name/] = [for (trans : Transition | step.inTransitions
         ) separator (' || ') after(' || ')] CC_[trans.name/][/for]([step.
         variable.name/] [for (trans : Transition | step.inTransitions)
         before ('&& ') separator (' && ') ]! CC_[trans.name/][/for]);
[/for]
```

Listing 16: Level action computation

```
if(!<transitions_fired>){
   //for every step <st>
   if(<st.variable.name>) {
   [st.actions(LevelActions)[0].variable.name/] =
   [st.actions(LevelActions)[0].expressionCondition.getCExpr()/] ;
   }
   //for all level actions associated to the step <st>
}
```

Listing 17: updating outputs or actions

```
if(! transitions_fired){
   //for every level action <act>
   if([act.variable.name/] != [act.variable.name/] +"_Old"){
   digitalPinWrite("pin_"+ [act.variable.name/],[act.variable.name/]);
   }
}
```

Listing 18: Duration of activity variables computation

```
if([aVariable.name/]){ [aVariable.name/]_duration ++; }
else { [aVariable.name/]_duration = 0 ; }
```

The functions <objet.getCExpr()> and <objet.getOldCExpr()> are used to invoke the GRAFCET expression parser, generating the corresponding C expressions."

ANNEX 3 : THE MAIN ACCELEO MODULE

The primary Acceleo module responsible for code generation is illustrated in Figure 8.

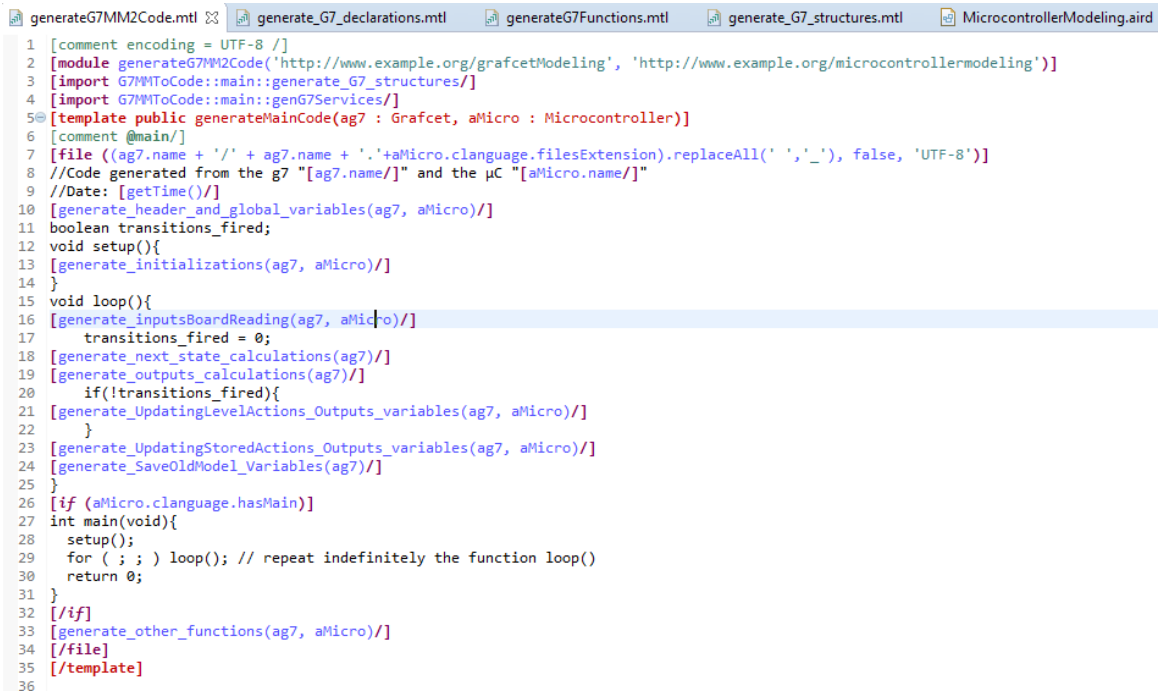

Figure 8: Overview of the main Acceleo module for code generation

ANNEX 4 : OVERVIEW OF THE ARDUINO CODE GENERATED FOR THE EXAMPLE

Listing 19: Overview of the Arduino code generated

```
#include "TimerOne.h"
//****  Declare INPUT pins mapped **** Total : 9
const byte pin_init_ = 2;
...
//****  Declare DIGITAL INPUT pins states **** Total : 9
boolean init_, init__Old;
...
const unsigned int TIMER_PERIOD = 100; //100 ms = 1/10 seconds
//Program Initialization
void setup(){
   initializeTimer();
   //INPUT PINs Configuration
   pinModeConfig(pin_init_, INPUT);
   pinModeConfig(pin_hT2, INPUT);
      ...
   //OUTPUT PINs Configuration
   pinModeConfig(pin_VR1, OUTPUT);
   pinModeConfig(pin_C, OUTPUT);
      ...
   //Inital steps activity variables initialization
   X1 = true; X2=false; X3=false; X4=false; X5=false; X6=false; X7=false;
      X8=false;
};
//Program loop
void loop(){
   //Reading states of Digital INPUT pins (Digital Input variables)
   init_ = digitalPinRead(pin_init_);
   hT2 = digitalPinRead(pin_hT2);
```

```
    ...
//Evaluate validated transitions (variables)
VT_1 = X1 ;
    ...
VT_6 = X4 && X5;
    ...
//Evaluate Receptivities of transitions
R_1 = (init__Old == false) && (init_ == true);
R_2 = hT2 && rain;
R_3 = ((! rain) && bWD) && ppM1;
    ...
R_6 = (X_4__duration >= 5000/PROGRAM_PERIOD) && (X_4__duration <= 10000/
    PROGRAM_PERIOD) && (tmp > 21) ;
...
//Evaluate clearing/firing transitions conditions
CC_1 = VT_1 && R_1;
CC_2 = VT_2 && R_2;
    ...
//Calculation if there is any transition cleared : 2nd alternative
transitions_fired = CC_1 || CC_2 || CC_3 || CC_4 || CC_5 || CC_6 || CC_7
     || CC_8 || CC_9 ;
    ...
//Evaluate steps activity variables
X1 = (X1 && ! CC_1);
X2 = CC_9 || CC_1 || CC_6 || (X2 && ! CC_4 && ! CC_3 && ! CC_2);
    ...
//Evaluate Digital OUTPUTs variables : 8
if(transitions_fired == false){
   //Evaluate Level Actions Associated to Step 3 : 1
   VR1 = X3 && (1);
   A = X6 && (bWD || ppM1) || X7 && (bWD && ! ppM2);
         ...
}
//Evaluate Analog/Stored OUTPUTs variables
//Evaluate Stored Actions Associated to Step 1
//Step 1: Action C On Activation
if(X1_Old == false && X1 == true){
   C = 0;
}
//Evaluate Stored Actions Associated to Step 3
//Step 3: Action C On deactivation
if(X3_Old == true && X3 == false){
   C = C + 2*N;
}
    ...
//Updating LEVEL ACTIONS OR DIGITAL OUTPUTs
if(!transitions_fired){
   //A stable situation is reached
   if(VR1_Old != VR1){
      digitalPinWrite(pin_VR1, VR1);
   }
   ...
}
    ...
// Keep the state of Xi variable in Xi_Old before the next cycle to use
    it when evaluating rising edge or falling edge of variables
X1_Old = X1;
...
```

```
}

void initializeTimer(){
   unsigned int FT_Steps = 1000/TIMER_PERIOD;
   Timer1.initialize(1000000/FT_Steps);
   Timer1.attachInterrupt(update_G7TimingVars_callback);
}
void update_G7TimingVars_callback(){
   //called periodically to update timing variables
   //Updating durations of steps activity variables for timing conditions
   //for the step 1
   if(X1){X1_duration ++;}e lse { X1_duration = 0; }
   ...
}
...
//Pin mode configuration
void pinModeConfig(int pin_num, int mode){
   pinMode(pin_num, mode);
}
...
```