Model-Driven Engineering Approach For SysML Activity Diagram Simulation

Damien Foures, Vincent Albert, Jean-Claude Pascal, Alexandre Nketsa
CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
University of Toulouse ; UPS; F-31077 Toulouse, France
{dfoures, valbert, jcp, alex}@laas.fr

Keywords: MDE, transformation, verification, SysML, OMG, ATL, Ecore, VHDL-AMS, TINA, Petri Nets

Abstract

This study aims to automate the simulation of activity diagram (AD) in accordance with the OMG SysML specifications. We use the concept of model-driven engineering to transform AD into VHDL-AMS. This transformation is depicted in two transformations based on specifications given by the Object Management Group (OMG): Activity Diagram (AD) to Petri net (PN) and PN to VHDL-AMS. We have established transformation rules in ATLAS Transformation Language (ATL). The semantic of Activity Diagram was expressed by LTL property and verified with the "model-checker" TIme petri Net Analyzer (TINA). The first transformation is used for formal verification. The second step allows to execute and simulate a system behaviour modelled by an AD. All simulations were implemented with SystemVision.

1. INTRODUCTION

1.1. Context

The present work is based on the general context of systems engineering, and integration of heterogeneous systems. These systems are embedded systems (software and hardware components), generally high real-time constraints and disciplines (electrical, mechanical, information, hydraulic...). We seek to propose methods and tools to assist the development cycle of such systems. The use of models and simulation is becoming dominant component in the development cycle, and we seek to improve (and eventually to automate) their use. The use of meta-modeling moves in this direction, as it aims to ensure cohesion between all products of the development cycle, considering that all is model. Based on the instantiation of a meta-model, the model description is clearer and less ambiguous.

1.2. Approach

This paper is based on concepts preconized by the OMG called Model Driven Architecture (MDA), itself based on modeling and automatic transformation of models into others models for simulation.

To develop a complete chain of transformation from activity diagram to simulation, we use TINA formalism [LAAS 2011], TINA toolbox [LAAS 2011], and a first transformation procedure from AD to PN [Foures et al. 2011]. TINA formalism allows us to verify formally that Activity Diagram properties are preserved by our equivalent PN, using its model-checking tool (selt).

A second transformation: PN to VHDL-AMS [Albert et al. 2005] allows us to propose a simulation phase, commonly called virtual prototyping. The addition of these two approaches allows us to validate the discrete and continuous part of the activity diagrams, and hence predict functional characteristics of the system.

Section 2 presents related works. In section 3 gives an introduce MDE (Model-Driven Engineering), in subsections we present briefly Activity Diagram, Petri Net and its metamodels. The fourth section show all transformation required to simulate AD. Section five present transformation verification. At the end of this paper, we presents one example, close to industrial environment, it is an abstract of complete study of fuel injection calculator. During this paper one running example:"the butterfly", illustrates each paragraph and equally show possibility of data management created by our solution.

2. RELATED WORKS

Several transformations from ADs have been developed. For example, in [Bonhomme et al. 2008], authors do not take into account many properties of ADs. For instance, the OMG specification say that tokens accumulated in a ControlFlow must be consumed when the action starts. This transformation does not handle such property then the PN behavior is not equivalent to the AD behavior. Using HiLeS may however be useful to keep the hierarchy of ADs. Neverless our approach aims to hide PNs implementation of model. We argue that our transformation is a strict interpretation of the OMG's SysML Activity Diagram operational semantic. As far as we know, this has not been done yet.

Stereotyping is common during transformations, it simplifies UML diagrams and outcome transformations. It's a good thing when you target a particular technical area. In our work, we use the initial SysML metamodel to establish transformation rules which gives a more general transformation than the one given by [Pllana et al. 2008]. Moreover, this latter work deal with UML 1.0 Activity Diagram which have been extended considerably by UML 2.0 or SysML. For instance, in our work we deal with *Continuous DataFlow* and *Pins*.

3. MODEL-DRIVEN ENGINEERING

Model-driven engineering (MDE), is one of few domains of computer science and system engineering, it provides tools, concepts and language to create and transform models. The increasing complexity of systems has led to model developers to handle higher-level concepts. The Object Management Group shown its interests in Model Driven Engineering and proposed in 1997, the standard MOF (Meta-Object Facility). This work uses the SysML Activity Diagram metamodel (MMAD) defined by OMG through the MOF. A metamodel is a definition of a given point of view of a modeling language. It is the model of the model. First, a model unlike any other model, depends on the system to model. It equally depends on the designer's system vision. According to designer's mastery of modeling language, the resulting model can be different. Yet, all these models are grouped under the same syntax and semantics. There is a language, a formalism which can be modeled itself. It is the basic principle of the meta-modeling.

3.1. Understanding the SysML MMAD

The activity diagram is one of the four behavioral diagrams included in SysML. They are useful to describe a hierarchical behavior, delayed, or a mixed systems.

The OMG specification says (Figure 1) An *Activity* is composed of *ActivityNode* and *ActivityEdge*. *ActivityNode* can be: a *ControlNode*, an *ObjectNode* or an *Action*. Each nodes is specialized because of associated arguments. Nodes may have really different behavior, so we will not present in detail each of the possibilities offered by SysML and prefer to refer to the OMG specification [OMG 2010b] .

More precised semantics is also given by the specification. For instance:

- For *ControlFlow* (page 401): Tokens offered by the source node are all offered to the target node.
- For *ActivityFinalNode* (page 339): An activity may have more than one activity final node. The first one reached stops all flows in the activity.

In this work, we attached specific attention to transcribe this precise semantic on PN structure. To illustrate it and all other steps on this paper, we will present now the "butterfly" example.

3.1.1. The Butterfly:

This simple example shows how to solve a differential equation of second order with two entities, each solving a differential equation of first order, and exchanging results with each others, after any iterations round of resolution.

Differential equations are:

$$\dot{q}_1 + 200.10^3 q_1 + 30.10^6 q_2 = 5.10^3 \quad (1)$$

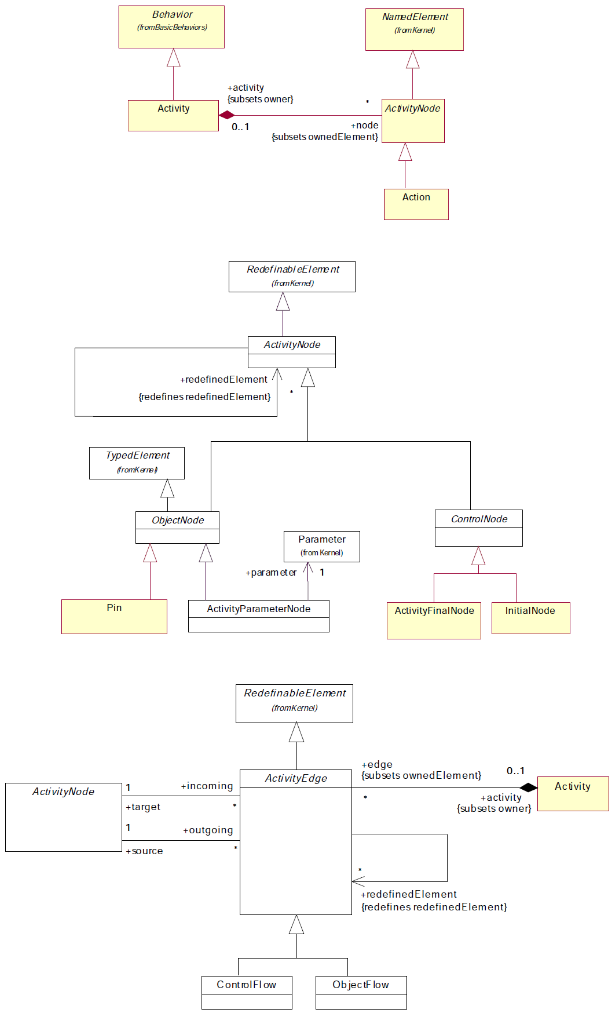


Figure 1. ActivityFinalNode meta-class

$$\dot{q}_2 - q_1 = 0 \quad (2)$$

Equation (1) (resp. (2)) is associated with the action I1 (resp. I2) of the activity diagram of Figure 2. When activity starts, two actions are ready to run, I1 is initialized and starts solving the associated equation. At each cycle of a given resolution data is provided through OF1 (ObjectFlow1) to I2. I2 can begin to make a resolution, and carry out resolution of the equation before giving a new value for I1.q2 through OF2. Through this cycle of two first order differential equations, we obtain an equivalent to second order differential equation. Between each cycle, the resolution is stopped and the value is stock in memory, it is a design choice. It is quite possible to solve the two equations together and update the variables q1 and q2 in the resolutions when they are available. Here, it is solving alternately to show the consideration of token presence to start the action.

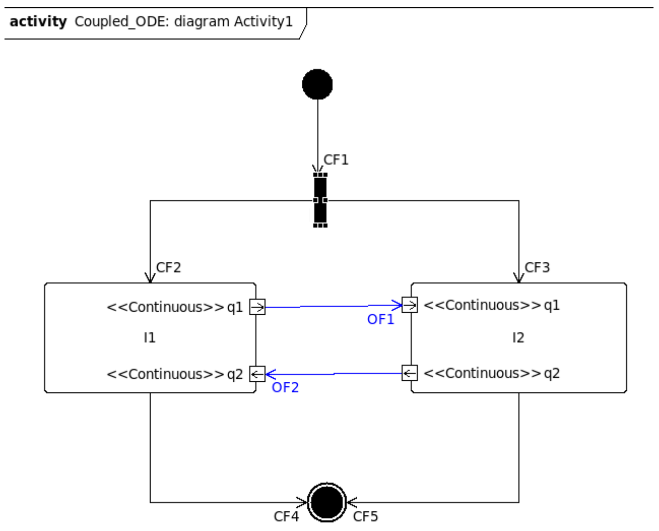


Figure 2. The butterfly activity diagram

At this step, the butterfly is not simulative, it must be transformed on PetriNet.

3.2. PETRINET PRESENTATION

A PetriNet is a mathematical modeling language. There are currently a lot of Petri nets classes. Gradually basic, hierarchical and differential predicate transition Petri nets, will be transformed to, the control part, the hierarchy, and finally the continuous part of the activity diagram. A Petri net is composed of places, transitions, and arcs. Arcs connect a place to a transition or a transition to place, others possibilities are forbidden. This kind of constraint must appear in the meta-model of PN.

3.2.1. PetriNet Meta-model

The PN meta-model established in [Albert et al. 2005] was adapted to this new work. Macro-place and macro-transition were removed because they are restrictive. For example, if an ActionNode of AD is transformed into macro-place it is impossible to put new value in this macroplace during execution. During execution macro-node becomes totally independant, so we decided to work flat. Flat PN, without hierarchy, are more easy to master communication links. TINA works also on only one abstraction level.

Figure 3 shows a simplified version of PN meta-model. We can read on it: *PetriNet* is composed of *Node* and *ArcClassic*. A node can be a *Transition* or a *Place* and they are linked with *ArcClassic*. A node can have multiple *incoming* or *outgoing* *ArcClassic*s. An *ArcClassic* can only have one *Source Node* and one *Target Node*. This interpretation includes a description of the previous paragraph. However, constraints do not appear, they must be expressed, for example in Object COnstraint Language (OCL) [OMG 2010a].

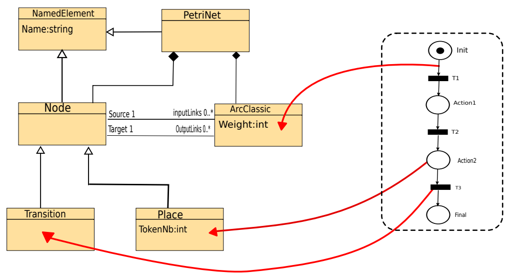


Figure 3. Petri Net model conforms to simple Petri Net meta-model.

4. TRANSFORMATION WITH ATL AND ECLIPSE MODELLING FRAMEWORK (EMF)

This work is based on transformation of activity diagram for simulation. Indeed, at what it seen to be a single transformation, its uses two transformations. There are two reasons, the first is historical, indeed transformation from Petri Net to VHDL-AMS [VHDL-AMS 1999] was already existing, the second is due to the part of Petri Net on V&V (Validation and Verification). In our approach, simulation for validation is regularly coupled to model-checking for verification. However it may be possible to deplay the same work with a unique transformation from activity diagram to VHDL-AMS.

4.1. Activity Diagram to Petri Net

Initially, our work was to be, totally in accordance with OMG. Tools for model transformation suggested by the OMG are still evolving, and to date we prefer to use EMF with Ecore meta meta-model and ATL language which seems to be the best choice, with a framework that has been already tried and tested. Our transformation choices are pointed out in figure 4.

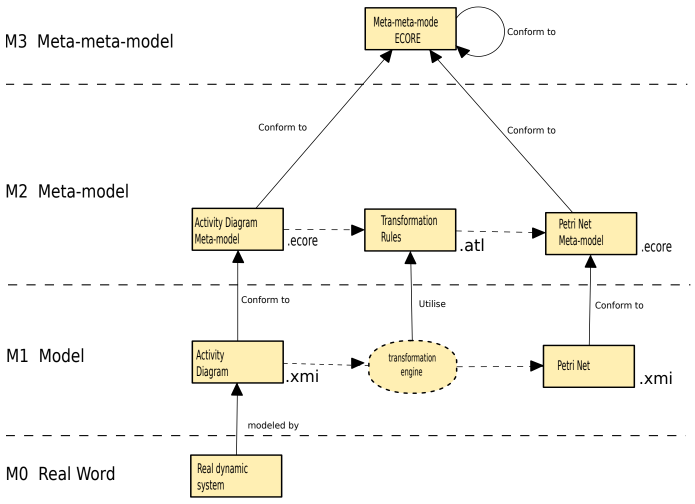


Figure 4. Meta-modelling Transformation.

4.1.1. Mapping of Concepts

The original contribution of our transformation is to match an activity diagram artefact to a PN block which will preserve the AD semantic, related to this artefact as defined by OMG. Such a PN block must also handle alternatives in AD modeling, e.g. an input pin may be stereotyped "optional" and becomes useless to start the activity. Table 5 illustrates the main mapping.

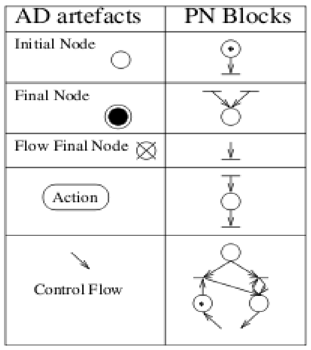


Figure 5. Basic Concepts Mapping: from AD to PN

These design choices, reflects the analysis based on the generality of blocks (SendSignalAction or CallBehaviorAction inherited from Action),on block interconnection facility but also on properties defined by [OMG 2010c] and [OMG 2010b]. For example, a ControlFlow can be modelised as a single transition [Thierry-Mieg and LHillah 2008], it can be also included on nodes like in [Bonhomme et al. 2008]. In figure 5 the ultimate PN block acts as a buffer to respect ControlFlow properties written in OMG specification. They define ControlFlow like an edge that starts an activity node after the previous one is finished, with this simple definition an PetriNet arc is a correct model, but OMG add many specification on ControlFlow or which influences behavior. Finally, PetriNet arc is inadequate to meet all properties. Let's look at an excerpt of the properties and define possible solutions to respect them:
Property 1 (from ControlFlow): *Tokens offered by the source node are all offered to the target node*.
Property 2 (from ActionNode): *When an action accepts the offer for control and object tokens, the tokens are removed from the original sources that offered them. If multiple control tokens are available on a single incoming control flow, they are all consumed.*
Solutions: We can do with the property 1 that the first intuition is good, PetriNet arc carry tokens too. Property 2 and many others shows that the ControlFlow has behavior of token storage like a PetriNet place. The inability to know dynamically the number of token in a place to empty correctly ControlFlow brings us to the model as a buffer. Indeed, the presence of token is important but not the token multiplicity.
The same work was done with almost every ActivityDiagram node. Many stereotypes can be applied to nodes and was not considered to date.

The transition from one column to another in figure 5 is possible at M2 level (see figure 4) with ATL rules and Eclipse Modeling Framework. We can see figure 6, the resulting petri net after AD2PN transformation. It is more complex in appearance, it takes the behavior of Activity Diagram, that part is not really readable but does not provide specific information aditional. To make this transformation, we must establish rules for each meta-class present through these instances in the model. For example: The ATL transformation rule for InitialNode meta-class.

```
rule initialnode_place{
from a:MMAD!InitialNode
to    b:MMH!Place (
   Name<-'p_Initial_'+a.name+'_'+a.activity.name,
   OutputLink<-c,
   ...
   ),
     c:MMH!ArcClassic(
   Name<-'a2_Initial_'+...
   ),
     d:MMH!Transition(
   OutputLink<-a.outgoing,
   Name<-...
   )
}
```

For each instance of InitialNode present in butterfly example (figure 2), a marked place is created, arc connects the latter to a transition (see figure 6). This transition is associated with meta-class instance after transformation, present in outgoing InitialNode argument (Outputlink←a.outgoing).

4.1.2. Complex Petri Net

It was already seen how to build an atomic block. Building complex PN is relatively simple, in an activity diagram every or almost every node are connected to another by "ControlFlow" or "ObjectFlow". They will just have to connect each atomic block (can be viewed as:Transition-Place-Transition) to controlflow or objectflow block (can be viewed as:Arc-Place-Arc). We remind the reader that, analysis at model level should be higher than meta-model level to establish the rules in MDE context. Using the hierarchy can significantly reduce the amount of transformation rules. With AD2PN transformation, we could see that ATL cannot use easily the advantages of hierarchy. The language must be well controlled to limit significantly the coding rules.

As an illustration, Figure 6 give the first generated PN by automatical transformation for butterfly example.

This graphical representation come from TINA toolbox. This graphical version under TINA is possible with a second transformation, from Petri Net to Tina (model2text).

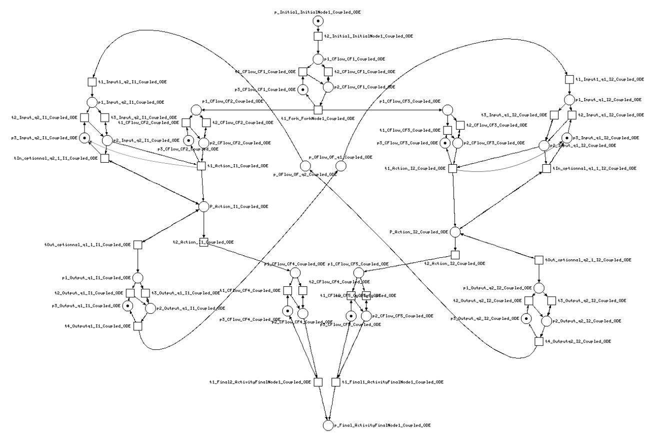


Figure 6. The butterfly Petri Net

The head of butterfly represents the initial node, CF1 and forkNode. Wing tips corresponds to continuous input and output pins. Wings represents CF2, CF3 and actions I1 and I2. The heart represents data exchange between OF1 and OF2. The tail represents CF4,CF5 and the final node.
As we can see a control flow is a relative complex PN structure allowing tokens accumulation and consumption

To make this transformation we have used ”Query”[LINA-INRIA 2006] from ATL :

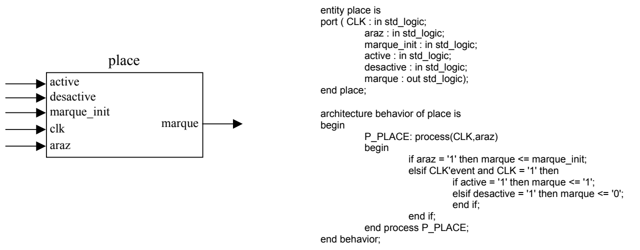
```
helper context Hiles!Transition def:genTransition():
String = ′tr ′ + self.Name + ′ [0,w[ ′
  + self.InputLink->iterate(arc;accPlAm:String=′ ′|
           accPlAm+arc.Source.Name +′ ′) + ′ ->
  + self.OutputLink->iterate(arc;accPlAv:String=
           accPlAv+arc.Target.Name +′ ′) + ′\n′
```

On this part of ”Query”, it is automatically generated the ”arc part” of tina text. Once adapted to an industrial scale this aspect of Petri Nets is meaningless, it is what brings us to perform automated formal verification. Petri Net formalism permits to verify automatically with LTL (Linear Temporal Logic) properties the good progress of transformation, but also user defined properties to verify part of the design which may be captured throught OCL contraints on the activity diagram. As we will see in part verification of this paper, all discrete behaviors of activity diagrams are taken into account. Currently, analog part was not present on Petri-Nets and it is directly implemented on VHDL-AMS files, futhermore TINA does not take into account hybrid PN. One file describe the discrete part of AD, the other describe the continuous part of AD.
The second transformation called ”PetriNet2VhdlAMS” also uses a MDE approach with ATL (ATLAS Transformation Language) a model2text transformation.

4.2. PetriNet to VHDL-AMS

We remind us, a Petri Net is a graph consisting of two types of nodes, places and transitions. Oriented arcs connect the places to the transitions. A marked Petri net contains marks or tokens distributed through the places. A place is therefore empty or marked. This distribution describes the discrete state of the model. We will focus here on two essential concepts of a Petri Net: the place artifact and the dynamic aspect of a Petri net.

Place component: The aim of the *place* component is to update the marking of the Petri net. This component implements a functioning which is either synchronous or asynchronous. Figure 7 below shows the interface (*entity*) and the body (*architecture*) of the synchronous place component.



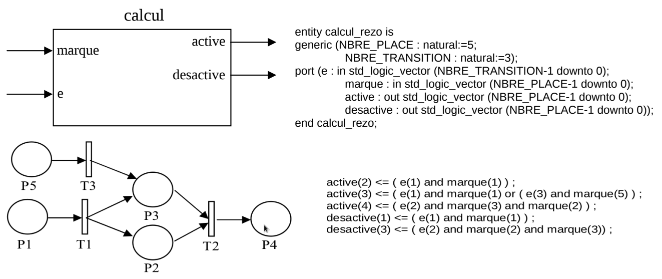
```
entity place is
port ( CLK : in std_logic;
        araz : in std_logic;
        marque_init : in std_logic;
        active : in std_logic;
        desactive : in std_logic;
        marque : out std_logic);
end place;

architecture behavior of place is
begin
        P_PLACE: process(CLK,araz)
        begin
                if araz = '1' then marque <= marque_init;
                elsif CLK'event and CLK = '1' then
                        if active = '1' then marque <= '1';
                        elsif desactive = '1' then marque <= '0';
                        end if;
                end if;
        end process P_PLACE;
end behavior;
```

Figure 7. Synchronous place component

Calcul component: The aim of the component *calcul* is to implement the dynamic aspect of a Petri net. It determines the evolution of the marking of the model *active* or *desactive* according to the sensitization *e* (conditions associated with the firing of the transition) and the current marking (*marque* of the net. In other terms, it determines the input/output flow of each place of the net.

The interface and the architecture of the component *calcul* is shown on Figure 8 below.



```
entity calcul_rezo is
generic (NBRE_PLACE : natural:=5;
          NBRE_TRANSITION : natural:=3);
port (e : in std_logic_vector (NBRE_TRANSITION-1 downto 0);
          marque : in std_logic_vector (NBRE_PLACE-1 downto 0);
          active : out std_logic_vector (NBRE_PLACE-1 downto 0);
          desactive : out std_logic_vector (NBRE_PLACE-1 downto 0));
end calcul_rezo;
```

```
active(2) <= ( e(1) and marque(1) ) ;
active(3) <= ( e(1) and marque(1) or ( e(3) and marque(5) ) ;
active(4) <= ( e(2) and marque(3) and marque(2) ) ;
desactive(1) <= ( e(1) and marque(1) ) ;
desactive(3) <= ( e(2) and marque(2) and marque(3)) ;
```

Figure 8. Calcul component

The dimensions of the input and output vectors are fixed dynamically according to the net structure. For this, we will

use generic parameters NBRE PLACE and NBRE TRANSITION which correspond to the number of places and number of transitions of the net respectively.
To create this component we use *Query* from ATL (Atlas Transformation Language) [LINA-INRIA 2006], this is model2text transformation.

```
helper context MMPN!Nbr def:genCalcul():
String = 'entity calcul_rezo is
generic (NBRE_PLACE : natural:='+self.nbr_Place.Name+';\n
NBRE_TRANSITION : natural:='+self.nbrTransition.Name+');\n
port (
e : in std_logic_vector (NBRE_TRANSITION-1 downto 0);\n
marque : in std_logic_vector (NBRE_PLACE-1 downto 0);\n
active : out std_logic_vector (NBRE_PLACE-1 downto 0);\n
desactive : out std_logic_vector ...) ;\n
end calcul_rezo;'\n
```

On this part of *Query*, it is automatically generated ”Calcul component”. The same type of *Query* is used to generate all ”Place component” and all needed components. This component and some others like ”allplaces” who connect each place to ”calcul component”, translate the discrete behavior of Petrinet, indirectly that of AD. This part can be created with classical VHDL files. VHDL does not include the management of the analog part, is the reason why we use VHDL-AMS (AMS for Analog and Mixed-Signal).
The principle of hybrid systems is to interact discrete aspects with continuous aspects of the system. The integration of the two views of this model is as follows: put a token in a place triggers the activation to the resolution of corresponding differential equations. Meanwhile, some numbers of thresholds are monitored. Each threshold being associated with a transition to a downstream marked site. When the threshold is crossed, the transition is crossed, a new marking is calculated (token value) and a new system of equations is activated.
This principle is directly inspired from management practices of Predicate/Transition Nets [Genrich 1987]. It is now possible to integrate equations (1) and (2) of the butterfly example in the simulation files, and launch their resolution when the marking of the PN is correct.

5. VERIFICATION

After establishing the rules for AD to PN, it is important to verify formally the transformation and, thus, verify that the PN had the same behavior as the activity diagram. In other words, it must check, through PN, to find the operational semantics of an AD. Subsequently, it is possible to imagine that users adds constraints (OCL) to the model, their validity in the PN can be proved with verification.

5.1. ResolveTemp Meta-model

Each PN block can be reduced to a sequence, Transition-Place-Transition. This meta-model defines each type of block to give essential features, but no behavior. It performs double transformation AD2PN and synchronized AD2ResolveTemp. This is to retrieve the name of input transitions (isStarted), output transitions (isFinished), running place (isRunning) and this incoming/outgoing (incomming/outgoing) (see figure 9).

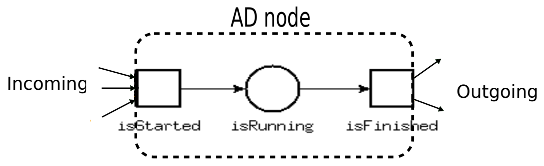


Figure 9. PN block definition

Sometimes attributes are added to define better LTL property (isNotRunning, optionnalIncoming,...)

5.2. LTL Properties and selt

Owing to lack of space, we will not present LTL language. Our approach has been to develop, properties in blocks with properties with inputs and outputs. In accordance with transitivity relationship A $\Rightarrow$ B and B $\Rightarrow$ C then A $\Rightarrow$ C. If block satisfies this properties, and if properties with connected blocks are satisfied then entire PN is verified. This verifies formally correct construction of the Petri net. This technique shows limitations indeed to have the expected Petri net (no problem in the construction). But it does not involve checking of correct behavior of the Petri net. If the building blocks have a limited or incorrect behavior, the model will be wrong and yet the verification will be positive. The user must know the limits of model transformation used. To generate automatically properties in LTL language, we use an other transformation: from ResolveTemp to LTL (model2text) and other ”Query”[LINA-INRIA 2006]:

```
helper context ResolveTemp!RTCF def:getPCF():String =
'[]('+self.isRunning+'+'+self.isNotRunning+'=1);\n'+...
```

The automated property created after this query is about ControlFlow and verifies invariant under block,the label '[]' means that this invariant must always be true to validate this property.

```
[](p2_CFlow_CF1_Activity1+p3_CFlow_CF1_Activity1= 1);
```

The OMG specification says:”*If multiple control tokens are available on a single incoming control flow, they are all consumed.*”[OMG 2010c]. To respect this semantic, controlflow is modelised as seen in concepts of mapping subsection (presence or exclusively absence of tokken in ControlFlow).

6. SIMULATION

The butterfly example is ready for simulation. We use *System Vision* of *Mentor Graphics* [SystemVision 2011] to simulate our VHDL-AMS files.

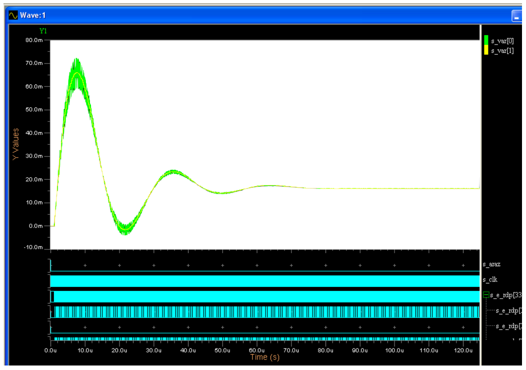


Figure 10. Simulation of butterfly example with *System Vision*

Finally, we obtain the simulation results in Figure 10. The discretisation effect comes from the structure example, newer values arrive synchronously to action I1 and I2. We observe oscillations of typical second-order equation with its passing before stabilization. This example shows that it is possible from an activity diagram, to automatically obtain an exploitable simulation thanks to formalism of PNs VHDL-AMS and even complex systems with a continuous exchange of data.

7. EXAMPLE

The previous example has shown a simple management of the data flow exchanges between action nodes. The application presented in this part is meant closer to industrial realities and was developed with an approach similar to that which can be found in research department. Its more complex structure brings up the concept of hierarchy and exchange data between hierarchical activities (*CallBehavior* concept in UML provides a way to interlock (imbricate) activities by activity invocation).

7.1. Specifications

The system is in charge of controlling the dosage of the gas mixture (air + fuel + gas exhaust) to provide for a 4-stoke engine (see figure 11).

The real-time system generates two types of information:

- A first discrete output calibrated on time (activation instant and width of activation) which controls the injection time (variable depending on the mode of the engine)
- A second type of continuous output controls the position of the recycling valve circulation for portion of the exhaust gases (to reduce pollution).

These two outputs are continuously generated from two sources of information:

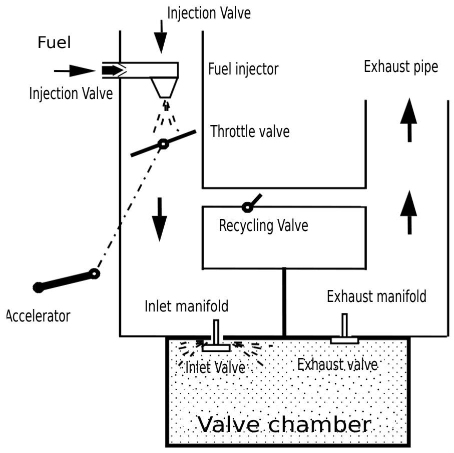


Figure 11. Schematic diagram of a fuel injection

- A set of measurements in real time using various sensors embedded dispersed in different parts of the engine.
- A set of parameters specific to each type of engine and car, determined through laboratory tests. They come in the form of tables and are provided as well to meet an economic goal (reducing the number and complexity of the sensors) and to reduce the technical complexity of the calculations in real time.

The Injection Control System three key factors:

- The quantity of the gas mixture air-fuel sent from the carburetor through the admission collector.
- The richness of the gas mixture ratio determined by air/fuel rate.
- The amount of exhaust gas recirculated into the collector admission.

The behavior of engine will be separated into three phases:

- Starting the engine.
- The increase in temperature of the engine.
- The operation at normal temperature, split itself into three phases: the engine is driven at constant speed, the engine is accelerating and the engine is decelerating.

7.2. From conception to simulation

According to each of these phases defined by sensors, the engine will receive a quantity of gasoline during a defined time. A detailed study of the system allows to obtain the context diagram, the diagram use case and sequence diagrams. We can then model the data exchange in Figure 12 of activity diagram with the complete Injection Control System.

Detail analysis: To show in detail all possibilities given by this transformation, observe part of the activity diagram and the associated simulation results. To do this, we will study the Admission part of activity diagram (Figure 13) and associated calculation phases.

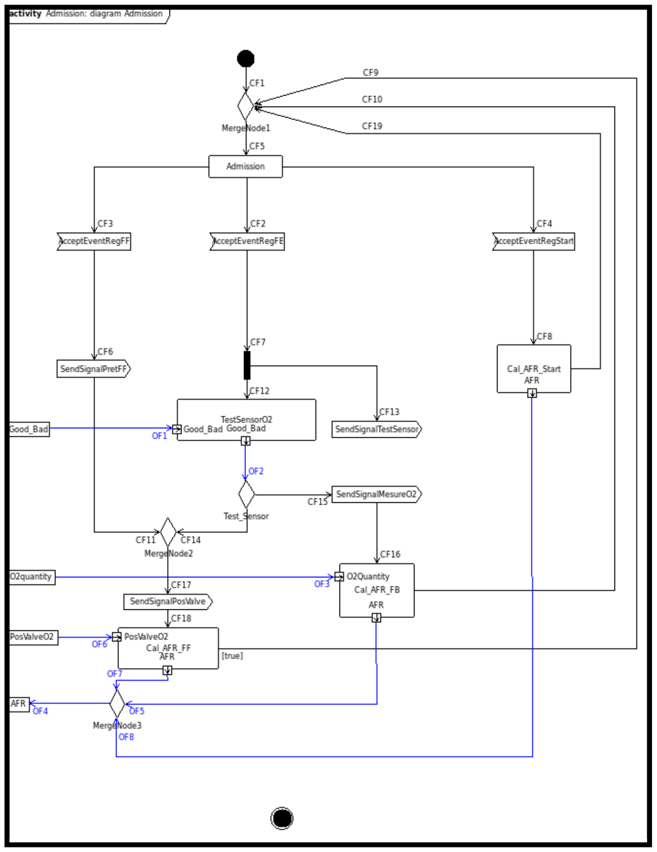


Figure 13. Admission activity of injection control system

The sensors send data to the engine part of the activity diagram, who chooses based on the number of revolution per minute of the engine and engine temperature one of three control modes: start control (cal_AFR_start) (AFR: Air/Fuel rate), feed-forward control (cal_AFR_ff), feedback control (cal_AFR_fb).

The scenario is such that, initially the engine starts-up (Step 1), when the number of revolutions per minute is less than 1000, then feed-forward control (Step 2), before reaching a sufficient engine speed arrive to feedback control (Step 3):

- Step 1: Sending a signal RegStar from Engine activity, received by the AcceptEventRegStart which launches start control (cal_AFR_start).
- Step 2: Sending a signal RegFF from Engine activity, received by the AcceptEventRegFF which launches feed-forward control (cal_AFR_ff).
- Step 3: Sending a signal RegFB from Engine activity, received by the AcceptEventRegFB which launches feedback control (cal_AFR_fb).

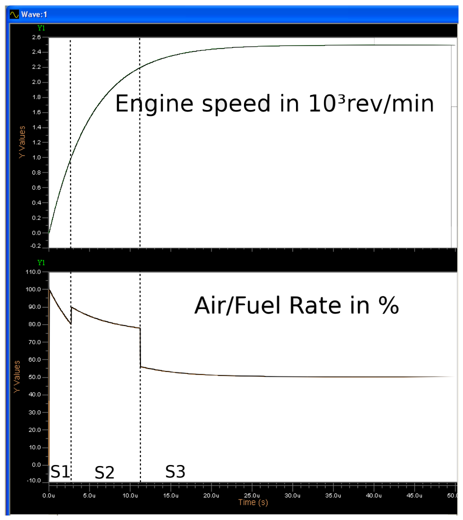


Figure 14. Simulation of Air/Fuel rate control

We find three phases in the simulation (Figure 14) with all three types of regulations and therefore different air/fuel rate depending on the speed (rev / min) of the engine. These rates are calculated by differential equations whose resolution begins the launch activities cal_AFR.

This example has shown that it is possible to transform Activity Diagrams into a formal language such as Petri Nets and be simulated. The combinatorial part of Petri nets is transparent for the user, it does not have to supervise the management of transitions from PN. The testbench is a way to control the progress of the simulation, but can remain transparent if no debugging is required, and of course it have not interaction with external stimulus.

8. CONCLUSION

The objective of this work was to propose a transformation of activity diagrams to VHDL-AMS in a MDE context according to the OMG specifications. The complexity of the OMG specification, the lack of maturity of tools implementing this specification is a significant barrier to the development of a solution in line with the OMG. Our objective to make this transformation as generic as possible to transform the majority of activity diagrams into VHDL-AMS files, which sometimes leads to serious solutions but transparent to the user. The simulation provides a vision of the signals carried in the activity diagram. The PN description of the AD is transparent to the user. The model can be self-governing (e.g.

without control of the environment). PN provides a mathematical formalism which is, at this step of the project not really exploited.

This study is functional and allows to develop transformation of a large part of activity diagrams but does not yet provide all the concepts present in the activity diagrams. A major work may establish rules respecting all OMG specifications. Full automation of the transformation, making Petri nets transparent to the user, and moving from the AD model to the simulation was the first step. We express concern about management of many changes which allows users (stereotype, optional attribute,...), to manage all of these cases seems to overload rules of transformation. ATL language has sometimes seemed a bit complex, it will be interesting to see the contribution of Query / View / Transformation (QVT) language. The VHDL code (for the discrete part) can be synthesized and embedded on a FPGA. Then, a further work would focus on an extension to co-design and hardware/software partitioning. While hardware part is transformed into VHDL, we wish to build another transformation to generate C-code for software part. Hardware/software partitioning should be identified in the earlier phases of the development cycle and may be capture with SysML using the concept of allocation.

REFERENCES

Albert V.; Nketsa A.; and Pascal J.C., 2005. *Towards a metal-model based approach for hierarchical Petri net transformations to VHDL. European Simulation and Modelling Conference, Porto*.

Bonhomme S.; Campo E.; Estève D.; and Guennec J., 2008. *Methodology and Tools for the Design and Verification of a Smart Management System for Home Comfort. 4th International IEEE Conference "Intelligent Systems"*.

Foures D.; Albert V.; and Pascal J.C., 2011. *ActivityDiagram2PetriNet: Transformation-based Model in accordance with the omg sysml specification. 25th European Simulation and Modelling Conference- ESM'2011, October 24-26, 2011, Guimaraes, Portugal*, , no. 1.0, 1–5.

Genrich H.J., 1987. *Predicate/Transition Nets.* In *Lecture Notes in Computer Science*.

LAAS, 2011. *http://homepages.laas.fr/bernard/tina/*.

LINA-INRIA A., 2006. *ATL:Atlas Transformation Language ATL User Manual. OMG specification*, , no. version 0.7.

OMG, 2010a. *OMG Object Constraint Language (OCL), Superstructure. OMG specification*, , no. 2.3, 1–256.

OMG, 2010b. *OMG Systems Modeling Language (OMG SysML). OMG specification*, , no. 1.2, 1–246.

OMG, 2010c. *OMG Unified Modeling Language(OMG UML), Superstructure. OMG specification*, , no. 2.3, 1–742.

Pllana S.; Benkner S.; Xhafa F.; and Barolli L., 2008. *Automatic Performance Model Transformation from UML to C++*. In *Parallel Processing*. 228 –235.

SystemVision M.G., 2011. *http://www.mentor.com/*.

Thierry-Mieg Y. and LHillah o.M., 2008. *UML behavioral consistency checking using instantiable Petri nets. ISSE*, 4, no. 3, 293–300.

VHDL-AMS, 1999. *Institute of Electrical and Electronics Engineers Standard VHDL Analog and Mixed-Signal Extensions.* , no. IEEE Std 1076.1-1999.

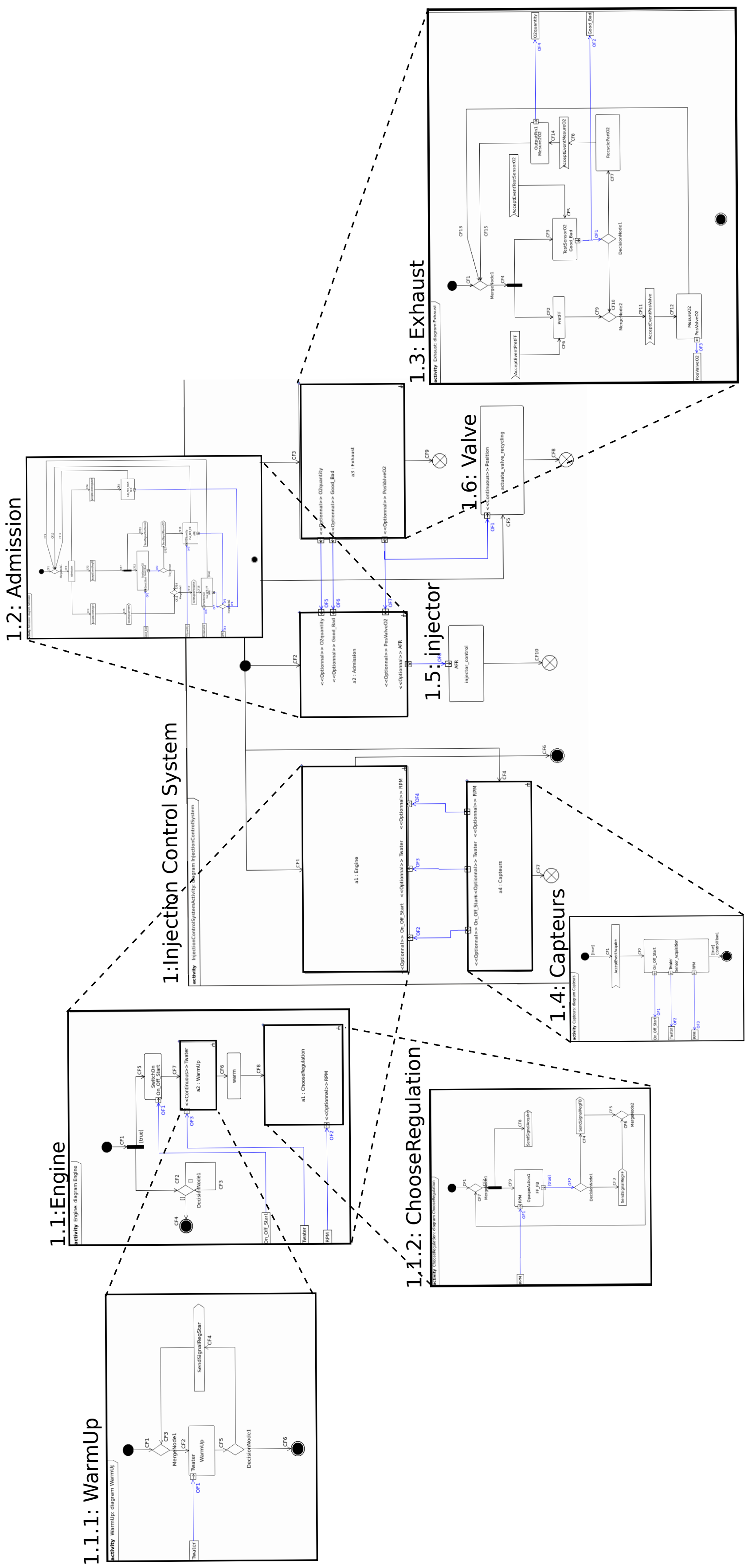


Figure 12. Activity diagram of Injection Control System