Contents

List of Tables

List of Figures

Introduction

Motivation

Unified Modeling language (UML) [1] is one of the most beneficial languages that becoming today a benchmark to specify the dynamic behavior of systems, it's used for modeling a several types of concurrent and complex systems like : aircraft control, hospital life support equipment, computer networks and banking systems...etc. For these types of systems it is essential that they work correctly from the very beginning, and to cope with that UML is insufficient because it's a semi-formal language, when it is crucial to provide a good method that enables to prove or disprove the correctness of system before using it. Among the methods which today are proof of a certain rigor in the verification of the critical systems is formal verification.
The main reason why we think today about transforming the semi-formal statecharts diagrams to formal Petri nets model to ensure the following points:

- Provide analysis tools for critical systems in the first design phases (conception and modeling),
- Obtain high levels of insurance valuation,
- Deal with the shortcomings of UML regarding the analysis and verification of properties, and this type of transformation is one of the proposed solutions,
- Integrate formal methods with oriented object conception methods to benefit from the efforts of both approaches.

Related Research Studies

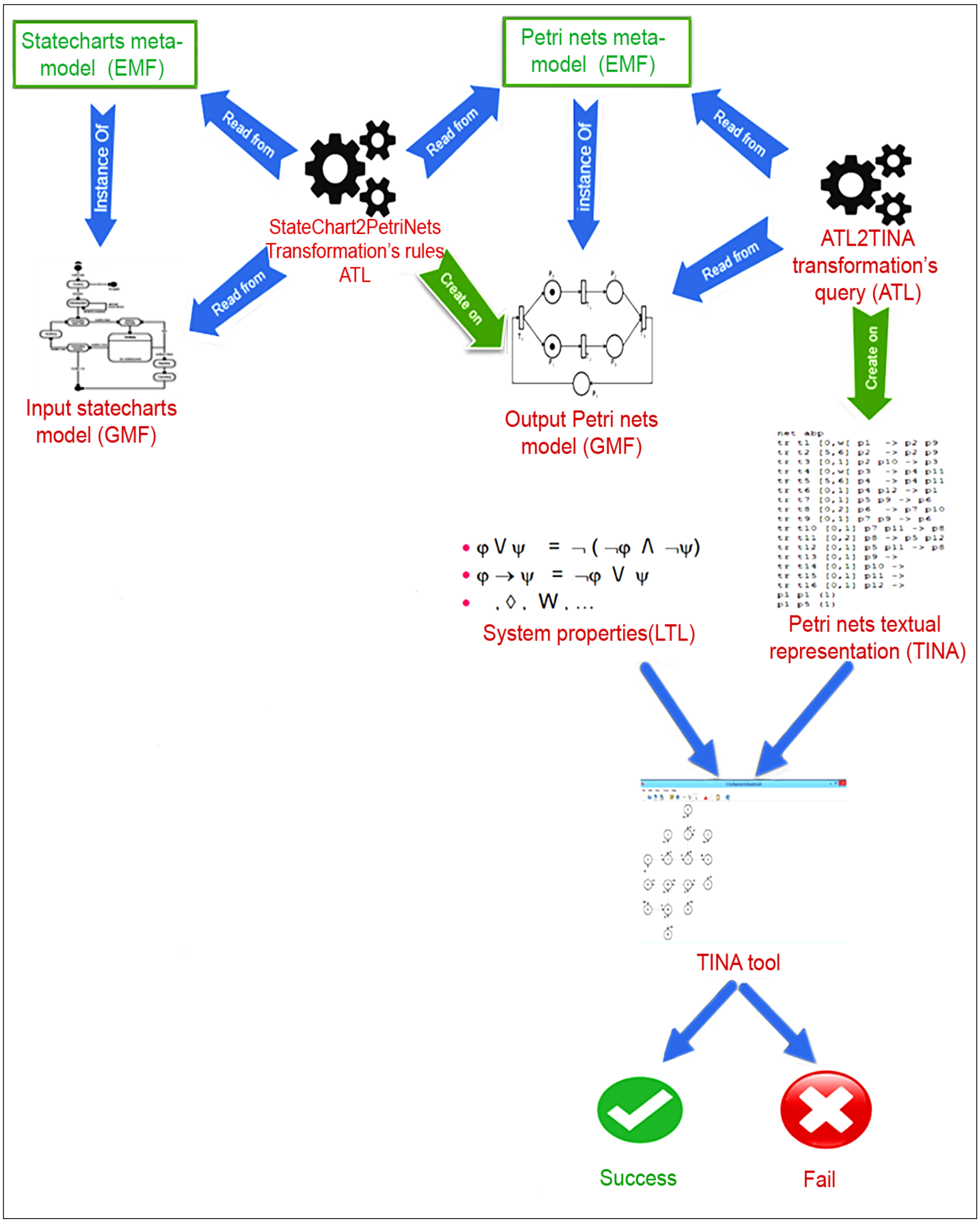
- Andrea Ambrogio [2] has proposed a framework of model transformation to automate the building of performance of UML models,

- A transformation from UML activity diagrams to stochastic Petri nets has been presented by Juan Pablo Lopez-Grao, Jose Merseguer, Javier Campos [3],
- L.Baresi, M Pezze [4] indicate the feasibility of ascribing formal semantics to UML by defining translation rules that automatically map UML specifications to high-level Petri nets,
- Oscar R. Ribeiro and Jo ao M. Fernandes [5] have proposed some rules to transform sequence diagrams into colored Petri nets,
- The same Job has been proposed, but there from activity diagram to colored petri nets by U. Farooq, C.P. Lam, and H. Li [6],
- Storrle, H [7] have proposed an approach to verifying data flow of UML 2.0 activities by transforming activity diagram to formal Petri nets model,
- M Bouarioua, S Chemaa, A Chaoui [8] have proposed a graph transformation based approach to analyze with TINA tool the result of the transformation from UML diagrams (statecharts diagrams) to stochastic Petri nets using a specific Java grammar libraries.

Contributions

The key technical contributions of this project include 4 fundamental steps :

- In the first step the statecharts and Petri nets meta-models has been created in order to instantiate any statecharts or Petri nets models,
- The second step represents the core of the transformation when we have created two grammars in which they contain 20 rules (grammar 1 : 7 rules, and grammar 2 : 13 rules) to transform any input state diagram to its equivalent on Petri nets using ATL,
- In the third step we will define the set of properties that we want to verify in our statechart system using LTL temporal logic,
- Finally TINA tool will be used in the formal verification step. It takes as input our resulted Petri net model and the set of properties expressed with LTL, and give as result if these properties has been satisfied in our formal Petri net model or not.



Statecharts meta-model (EMF)
Petri nets meta-model (EMF)
Instance Of
Read from
Read from
Instance Of
Read from
StateChart2PetriNets Transformation's rules ATL
ATL2TINA transformation's query (ATL)
Read from
Create on
Read from
Create on
Input statecharts model (GMF)
Output Petri nets model (GMF)
• φ V ψ = ¬ (¬φ Λ ¬ψ)
• φ → ψ = ¬φ V ψ
• ◻ , ◊ , W , …
System properties(LTL)
Petri nets textual representation (TINA)
TINA tool
Success
Fail

```
net abp
tr t1 [0,w[ p1  -> p2 p9
tr t2 [5,6] p2  -> p2 p9
tr t3 [0,1] p2 p10 -> p3
tr t4 [0,w[ p3  -> p4 p11
tr t5 [5,6] p4  -> p4 p11
tr t6 [0,1] p4 p12 -> p1
tr t7 [0,1] p5 p9 -> p6
tr t8 [0,2] p6  -> p7 p10
tr t9 [0,1] p7 p9 -> p6
tr t10 [0,1] p7 p11 -> p8
tr t11 [0,2] p8 -> p5 p12
tr t12 [0,1] p5 p11 -> p8
tr t13 [0,1] p9 ->
tr t14 [0,1] p10 ->
tr t15 [0,1] p11 ->
tr t16 [0,1] p12 ->
pl p1 (1)
pl p5 (1)
```

Figure : Project steps.

Chapter 1

Statecharts diagrams

1.1 Introduction

Unified Modeling Language (UML) [1] is a general purpose, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of any system. It was originally motivated by the desire to standardize the disparate notational systems and approaches to software design developed by **Grady Booch**, **Ivar Jacobson** and **James Rumbaugh** at Rational Software in 1994, with further development led by them through 1996.
In 1997 UML was adopted as a standard by the Object Management Group (OMG), and has been managed by this organization ever since. In 2005 UML was also published by the International Organization for Standardization (ISO) as an approved ISO standard.

1.2 Classification of UML 2.x diagrams

UML provides a several kinds of diagrams, which allow different aspects and properties of a system to be expressed, it has a several characteristics and provides the possibility to modulate each system with a different models and a different views. It's composed of 15 diagrams (Version 2.5), divided into two big parts : structural and behavioral diagrams, as shown in Figure 1.1.

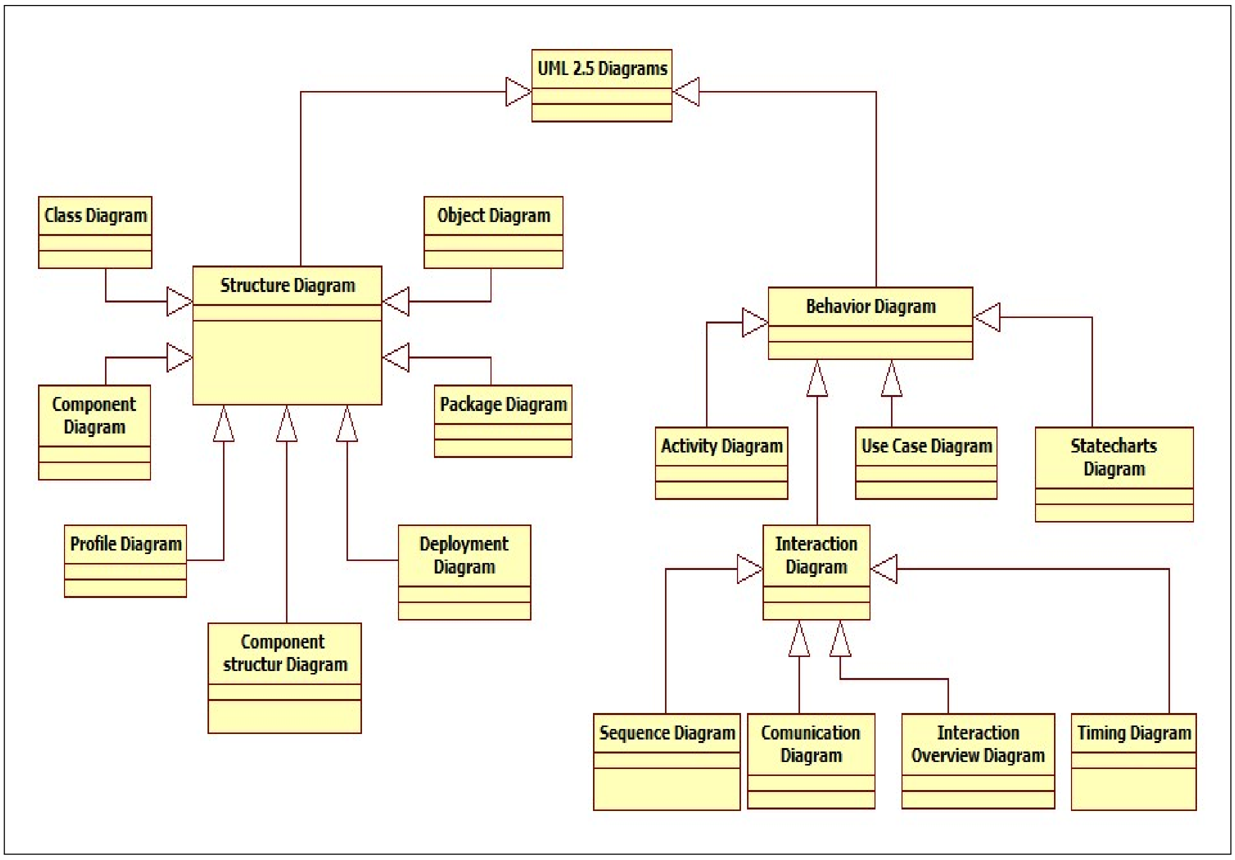


Figure 1.1: UML structure.

1.2.1 Structural diagrams

Are the most important and the widely used part of UML, used to express your understanding of the real-world problem, define the meaning of words you use in your proposed solution and to represents the static view of the system which can be designed using : classes, interfaces, components, nodes ...etc.

1.2.2 Behavioral diagrams

It shows us what should happen in a system, and describes how the objects can **interact** with each other to create a **functioning** system using a several types of elements like : messages, states, transitions, fragments ...etc.

1.3 Statecharts Diagrams

In this project we will set the point on one of the diagrams that taking part of the behavior UML side, it's statecharts diagrams .

Statecharts introduced originally by David Harel [9] are a finite state machine allowing a complete system to be expressed in a more compact and elegant way, it specifies the sequence of states that an object goes through during its lifetime in response to events.

1.4 Basic statecharts diagrams symbols and notations

David Harel has defined a set of elements and notation in order to represent a statecharts diagrams [10] such as: state, transition, action, event ...etc.

1.4.1 State

One of the main notions of statecharts is the state. It represents situation during the life of an object. You can easily illustrate a state in smartdraw by using a rectangle with rounded corners.

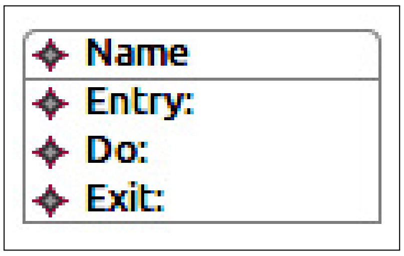


Figure 1.2: State.

Each state has 4 main parts :

- Name : It represents the name of the state,
- Entry : Which represents the action that will be executed once accessing the state,
- Do : It means execute *'DO'* activity while the object is in the state (but only after the state entry behavior has completed),
- Exit : Once quitting the state, execute the exit action.

In statecharts diagrams the following kinds of states are distinguished:

- Simple State: Every state in which it has no internal vertices or transitions is called **simple state**,

- Composite State: A composite state contains at least one region, and each region represents the lifecyle of an independent object.

1.4.2 Transition

A transition is a single directed arc, originating from a single source node and terminating on a single target node (the source and target may be the same node), each transition is labeled by an **event** that triggered it, and an **action** that results from this event.
The transition will be reached when its state machine execution has reached its source state(i.e., its source state is in the active state configuration), then it will be traversed once it being executed, and finally completed after reaching its target node.

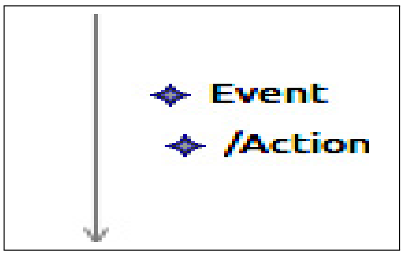


Figure 1.3: Transition.

In statecharts diagrams we have a several types of event :

- Signal events,
- Call events,
- Change events,
- Temporal events.

An action represents a specific method or operation that will be executed once the transition is riched.

1.4.3 Initial state

A filled circle followed by an arrow represents the object's initial state.

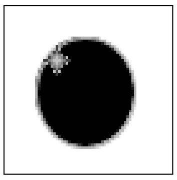


Figure 1.4: Initial state.

1.4.4 Final state

An arrow pointing to a filled circle nested inside another circle represents the object's final state and the end of an object's existence, in a composite state final node signifying that the enclosing region has completed.

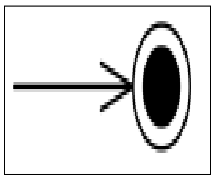


Figure 1.5: Final state.

1.4.5 Synchronization and splitting of control

OMG in UML version 2.5 has been added to the statecharts diagrams some elements to represent the synchronization and the splitting of control like: fork node, join node...etc.

- **Fork node**
  It is a control node that has one incoming edge and multiple outgoing edges. Is used to **split** incoming flow into multiple concurrent flows.

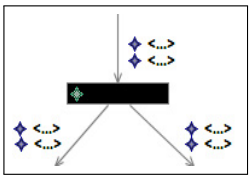


Figure 1.6: Fork node.

- **Join node**
  It is a control node that has multiple incoming edges and one outgoing edge and is used to **synchronize** incoming concurrent flows.

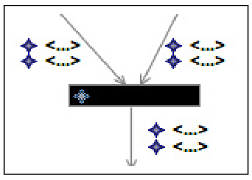


Figure 1.7: Join node.

1.4.6 History state

Once the composite state will be accessed, the history state is used to get the last active state registered before quitting the composite state.
Two types of history states are provided : **deep history** and **shallow history**, the distinction between them handles the case when the last state returned to is a composite state.

- Shallow history state: represents a return to only the **top most** substate of the most recent state configuration,
- Deep history state: represents the full state configuration of the **most** recent visit to the containing region.

Example:

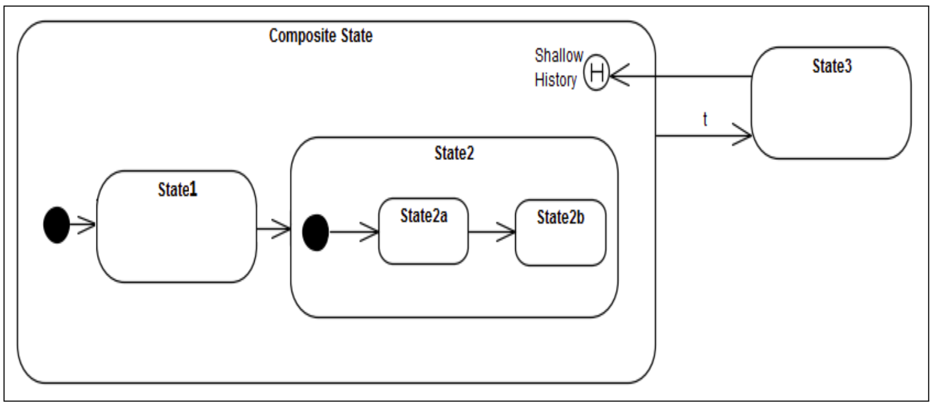


Figure 1.8: Shallow history state.

In Figure 1.8, with the shallow history we can get the last state that was active which can be the state 1 or state 2. However if the last active state is **state2a** or **state2b** shallow history cannot access them because they are not in the same level as shallow history pseudo state, and to deal with that we shall use deep history pseudo state as shown in Figure 1.9.

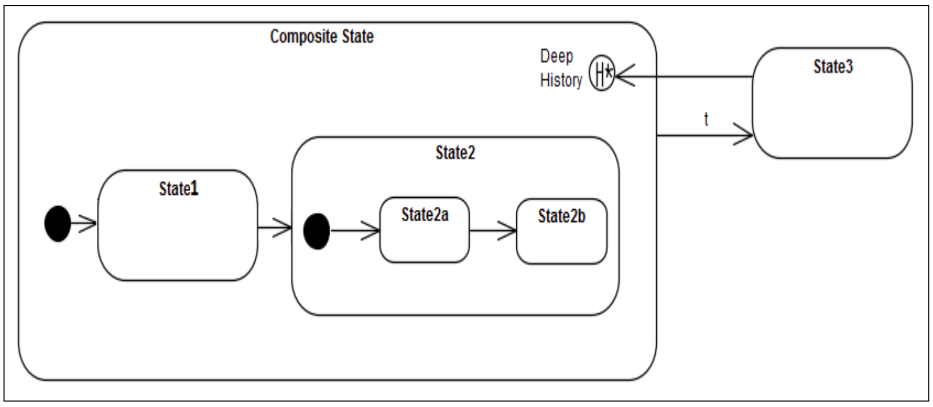


Figure 1.9: Deep history state.

1.5 Conclusion

This chapter has briefly presented the structure of the last version of UML the standard modeling language of the OMG which is 2.5 (released on 2015), by setting the point on one of the most important diagrams to representing the behavior of objects. Furthermore, we have presented the main elements and notations semantics of statechart diagram like : states, transitions, actions, events...etc.
Now to verify the behavior and properties of each system represented within this kind of diagrams, statecharts cannot offer a formalism allowing that, the reason why we call today UML as a semi-formal language. However models established in many mathematical domains like Petri nets, transitions systems, process algebras are precise and could be analyzed and verified by using various tools in these domains.
In the next chapter we will present Petri nets model, with the goal to transform next, our semi-formal statecharts models into Petri nets formalism in order to be able to verify our properties using a formal verification techniques.

Chapter 2

Petri nets models

2.1 Introduction

Petri nets [11] are a graphical and mathematical tool applicable to many systems. They are a promising tool for describing a studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel.
As a graphical tool, Petri nets can be used as a visual communication aid similar to flow charts, bloc diagrams and networks, in additions **tokens** are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool it is possible to set up state equations, algebraic equations and other mathematical models governing the behavior of systems [12].

2.2 Origin and domains of application

Historically speaking, the concept of Petri nets has its origin in Carl Adam Petri's dissertation [13], submitted in 1962 to the faculty of Mathematics and Physics at the technical university of Darmstadt, West Germany.
We focus on three application domains of petri nets: manufacturing, telecommunication, and workflow management [14].

- **Manufacturing :** Factory automation is probably one of the oldest application domains of Petri nets theory. Since the early 70s. Flexible manufacturing systems (FMS) in particular appear to be an interesting area of application. These systems are characterized by flexible, concurrently operating and mainly automated elements, such as a production controller, a machine, an automated guided vehicle and a conveyor. This results in high productivity, short throughput times and a high degree of diversity in output(i.e.the resulting products).

There are several reasons for using Petri nets in the domain of manufacturing. Petri nets allow for the modeling of resource sharing, conflicts, mutual exclusion, concurrency, and non-determinism. Moreover, the well-defined semantics allows for both qualitative and quantitative analysis. In particular, Petri net theory can help to detect potential deadlocks and construct control policies for deadlock prevention.

- **Workflow management:** The main purpose of a workflow management system is the support of the definition, execution, registration and control of processes. Because processes area dominant factor in workflow management, it is important to use an established framework for modeling and analyzing workflow processes and Petri nets are a good candidate for the foundation of a unified workflow theory.

- **Telecommunications:** A telecommunications system consists of two subsystems: a transport subsystem and a processing subsystem. The transport subsystem is the network (i.e.the communication resources). The processing subsystem is the set of computing resources and programs that control and manage the transport network on the one hand, and that implement the communication software on the other hand. The complexity of the two subsystems and the interaction between them may lead to design errors and performance problems. Therefore, a rigorous approach for modeling and analysis needs to be implemented.
  And there are several other application domains where Petri nets have turned out to be a useful design/analysis tool:

- Distributed software systems,

- Multi-processor systems,

- Asynchronous circuits,

- Distributed protocols,

- Hardware and software architectures,

- User Interfaces...etc.

2.3 Basic concepts

Petri nets are composed from a specific elements to describe any system like: places, transitions and arcs, and use the concept of marking in order to represent the behavior of these systems.

2.3.1 Place, transition and arc

A Petri nets model is a graph represented using a three different types of elements: circles (called places), bars (called transitions) and lines which links places and transitions (called arcs).

- **Place:** It represents the possible states of the system or the condition that should be verified to access the next transition.

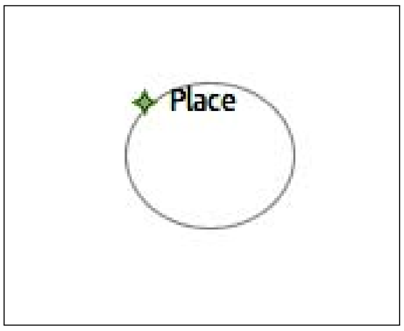


Figure 2.1: Place.

- **Transition:** It represents an action or an event.

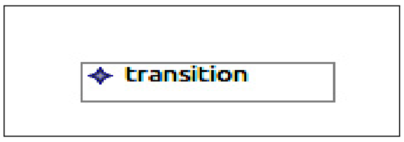


Figure 2.2: Transition.

- **Arc :** Every oriented arc simply connects a place with a transition or a transition with a place.

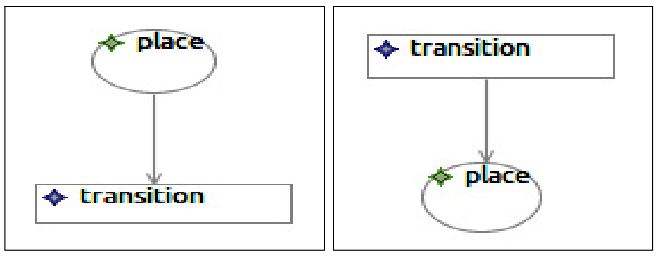


Figure 2.3: Arcs.

- **Token:** Each Petri net model is denoted by a movement of token(s) (black dots) from place(s) to place(s), and is caused by the firing of a transition.

2.3.2 Marking

In Petri nets each place contains an integer (positive or zero) number of **tokens** or **marks** contained in place P_i will be called m(P_i) , the marking defines the states of the Petri nets, or more precisely the state of the system described by the Petri nets. The evolution of the state thus corresponds to an evolution of the marking which is caused by firing of transition.

2.3.3 Firing of a transition

A transition can only be fired if each of the input places of this transition contains at least one token [17], the transition is then said to be **firable** or **enabled**. A source transition is thus always enabled. The transitions t_1 in Figures 2.4 (a), (b) and (c) (before firing) are enabled because in each case place P_1 and P_2 contain at least one token. This is not the case for (d) in which the transition t_1 is not enabled, since P_1 does not contain any tokens.
Firing transition t_i consists of with drawing a token from each of the input places of the transition t_i and of adding a token of to each of the output places of this transition. And this is illustrated in figure 2.4 (b) [15].

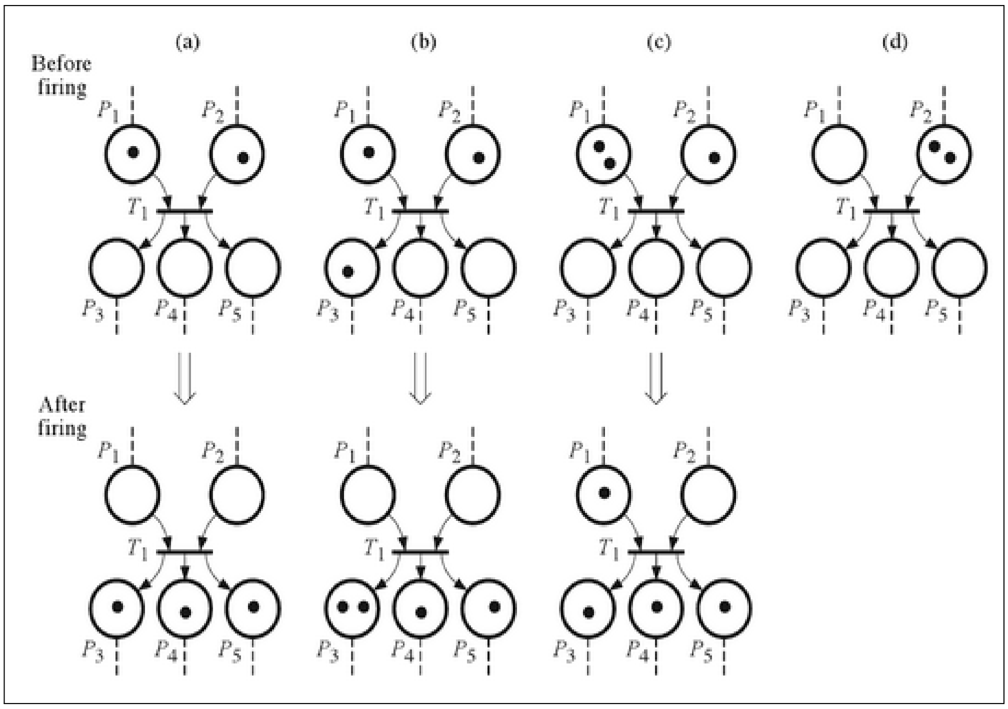


Figure 2.4: Firing of a transition.

2.4 Particular structures

Petri nets give the possibility to represent the concurrent and the synchronous systems using some particular structures, such as : sequencing, parallelism, resource shairing...etc.

2.4.1 Sequence a;b

Sequence of transitions in Petri nets, means that each of them will be firable one after one.

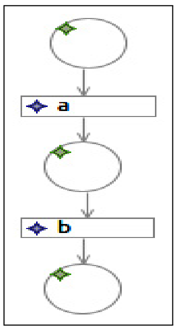


Figure 2.5: Sequence a;b.

2.4.2 Resource shairing

Resource sharing occur when a several process share the same resource in the same system, with Petri nets we model this situation as illustrated in Figure 2.6.
The resource is represented with a token (number 1 in the Figure 2.6 means one token), and each of processes p1 or p2 will take the resource only when it's available in place *p0* once firing the transition *t1* or *t3* with order.
When the process *p1* (*p2*) enable the transition *t2* (*t4*) it will make the resource in place *p0*.

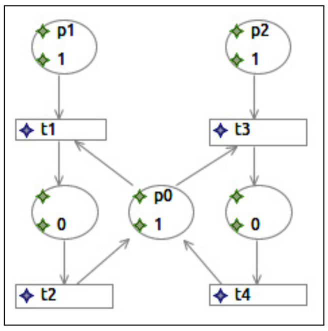


Figure 2.6: Resource sharing.

2.4.3 Parallelism

Parallelism with Petri nets is represented using a transition with a several exiting arcs.

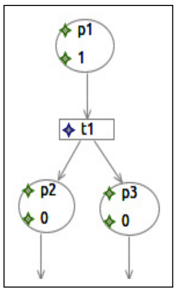


Figure 2.7: Parallelism.

2.4.4 Synchronization

There is two Kind : Mutual exclusion and Synchronization using signals.

- **Mutual synchronization**
  Used to synchronize the operations of many processes, for example in Figure

2.4, places *p1* and *p2* corresponds on processes *p1* and *p2* with order, transition *t* here cannot be firable until places $p1$ and *p2* having one token in the same time, and that can be used to represent some situations like an appointment.

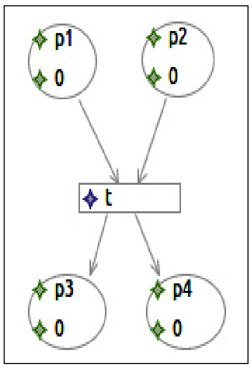


Figure 2.8: Mutual synchronization.

- **Synchronization using signals**
  Used to express communication between processes using signals, for example in Figure 2.5 process *p2* in wait and cannot fire the transition *t2* until receiving a signal from the process *p1*.

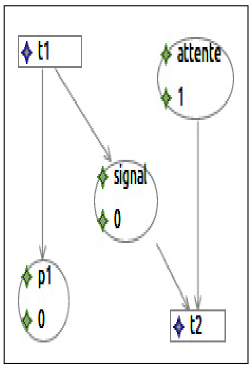


Figure 2.9: Synchronization using signals.

2.5 Types of Petri Nets

In some cases Petri nets using only places, transitions are not capable to express all the properties that we would like modeling, for this reason a several types of Petri nets are distinguished :

- Generelized Stochastic Petri nets [16]: a stochastic Petri net is a Petri net in which weight are assigned to arcs and represented with positive numbers, used to complex flow management systems,
- Temporal Petri nets [17]: it's a temporal extension of Petri net used for modeling time management in systems in which they have a big constrains of time,
- Hight Level Petri nets [18]: called also Colored Petri nets used for modeling collaboration system when there is much resource, because colored Petri nets offer a several kind of tokens, when each of them has a special color.

2.6 Conclusion

This chapter has presented the semi formalization of a powerful mathematical tool used as a formal modeling language for the critical system which is Petri nets model, and its basic components such as: places, transitions and arcs. Furthermore, the concept of marking and the mechanism of firing of transitions.
In the next chapter we will explain the main concepts of model transformation, with the goal to use them lately for implementing the transformation from statecharts diargams to Petri nets.

Chapter 3

Model transformation : Statecharts and Petri nets metamodels

3.1 Introduction

This chapter explains the main followed steps in order to create the statecharts and Petri nets editors based on its meta-models. But before that, we will take a look on how the concept of model transformation been achieved, and what are its main goals.

3.2 Model transformation's overview

What is OMG ? What is MDA ? What is MDE ? And how are these concepts are related ? It's really crucial to see that before understanding what the concept of model transformation means and how we are implemented the transformation from statecharts to Petri nets models.

3.2.1 OMG, MDA and MDE

In November 2000, the **OMG** (Object Management Group) made public the **MDA** (Model Driven Architecture) initiative, a particular variant of a new global trend called **model engineering**. The basic ideas of model engineering are germane to many other approaches such as generative programming, domain specific languages, model-integrated computing, software factories, ...etc. MDA may be defined as the realization of model engineering principles around a set of OMG standards like MOF (Meta Object Facility), XMI (XML Metadata Interchange), OCL (Object Constraint Language), UML (Unified Modeling Language)...etc. Similarly to the basic principle 'Everything is object' that was important in the 80s to set up the object-oriented technology, that the basic principle in model engeeniering 'Everything is model' may be key to identifying the essential characteristics of this new trend (see Figures 3.1 and 3.2) [19].

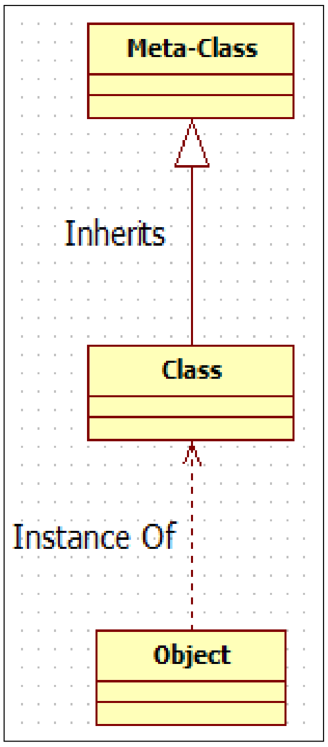


Figure 3.1: Object approach.

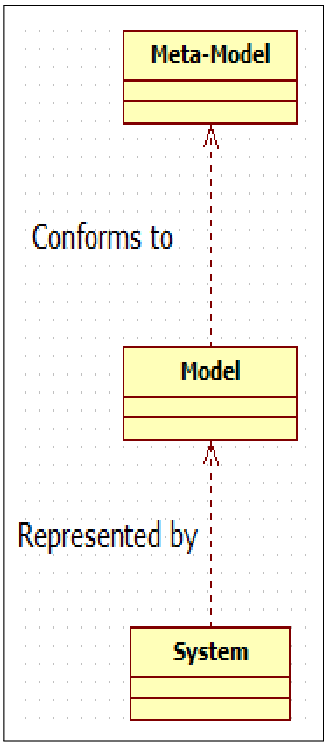


Figure 3.2: Model approach.

One of the main concept of model engeeniering is model transformation.
So model transformation is currently undergoing for defining some sort of Unified Model Transformation Language. This will allow transforming model *Ma* into another model *Mb*, irrespective of the fact that their corresponding meta-models *MMa* and *MMb* are identical or different. Furthermore, the transformation program is composed of a set of rules, should itself be considered as a model *Mt*. As a consequence, it will be based on the meta-model *MMt*. An abstract definition for this unified model transformation language is described in Figure 3.3.

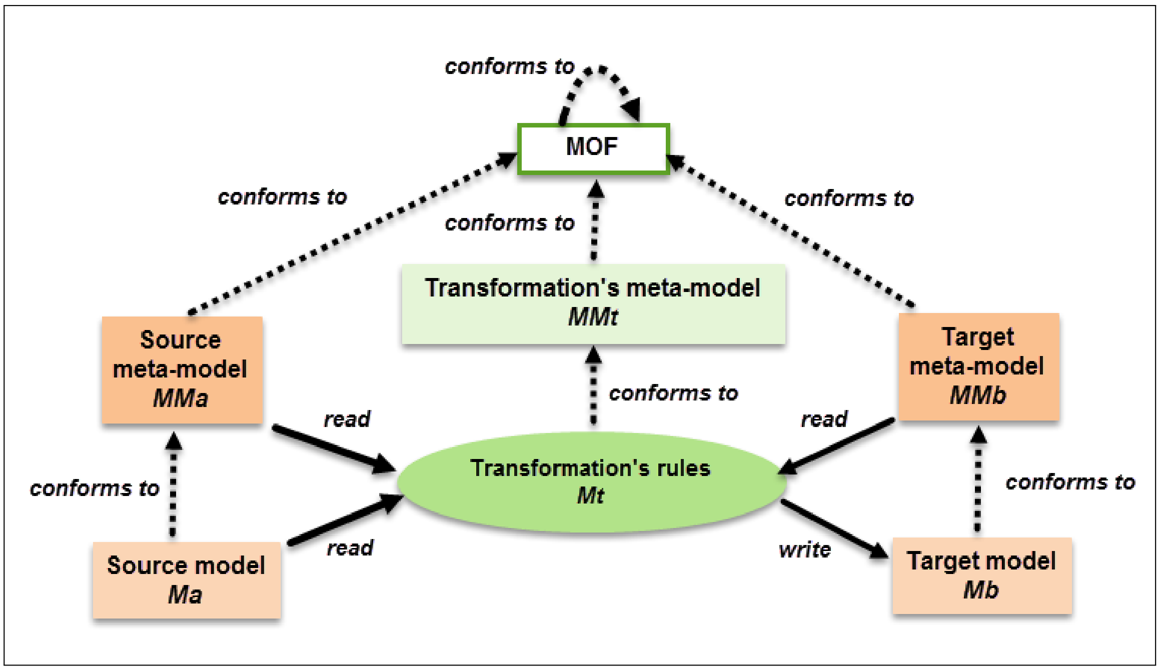


Figure 3.3: Model transformation.

Why we need a meta-model ?

The basic use of meta-model is that it facilitates the separation of concerns. When dealing with a given system, one may observe and work with different models of the same system, each one characterized by a given meta-model. In model transformation we use the meta-model concept to define the domain of the source and the target model in order to create a common grammar of rules to transform any source model conforms to its meta-model to its equivalent in a target model.
OMG is based on these concepts to define its 3+1 architecture shown in Figure 3.4.

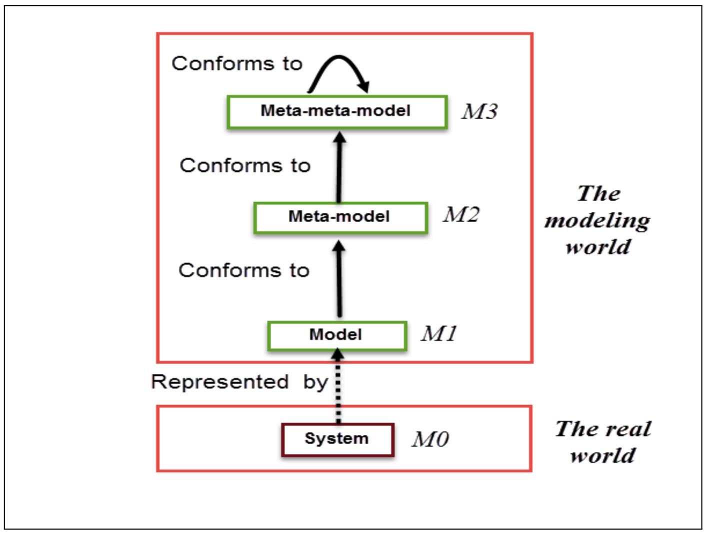


Figure 3.4: 3+1 MDA architecture levels.

3.2.2 Model transformation's approaches

Krzysztof Czarnecki and Simon Helsen in [20] have distinguished between two type of model transformation's approaches : model to code and model to model approaches.

Model to code approaches
They have distinguished between visitor-based and template-based approaches.

- **Visitor based approach :** A very basic code generation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream. An example of this approach is Jamda, which is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (so called CodeWriters) to generate code.

- **Template based approach:** Is a generative technology that transform data into structured text, e.g. code, through the use of templates. The data, referred to as the model, represents an abstract design while the templates provide the model of the target code used by the generator in the process of code production. The majority of currently available MDA tools support template-based model-to-code generation,e.g., b+m Generator Framework [30], AndroMDA [31], ArcStyler [32]...etc. (the detail of this approache is presented in [21]).

Model to model approaches
They have distinguished between : Direct manipulation, relational, graph transformation and hybrid approachs

- **Direct manipulation approach:** This approach offer an internal model representation plus some API to manipulate it. They are usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations (e.g., abstract class for transformations). However, users have to implement transformation rules and scheduling mostly from scratch using a programming language such as Java. Examples of this approach include Jamda and implementing transformations directly against some MOF compliant API (e.g., JMI) [20].

- **Relational approach:** This approach uses constraints to specify the relationships between elements of the source model and those of the target model using logic based on mathematical relations.
- **Graph transformation based approach:** It is the most used and natural approach, because the model itself can be considered as a graph. This approach is based on graph grammars defined by Chomsky. The process of transformation is based on the execution of a set of rules, each rules composed of two parts LHS (Left Hand Side) and RHS (Right Hand Side). The execution of each rule consist to replace LHS on the source model to its equivalents RHS, and finally after executing all the rules the final target model will be completely generated. The most popular tools based on the transformation graph are : VIATRA [22], AToM^3 [23], AGG [24], GreAT [26]...etc.
- **Hybrid approach :** Hybrid approach combine different techniques from the previous categories like ATL.

3.2.3 Model transformation's tools

A number of specialized languages has been proposed, aimed at specifying model transformation such as : VIATRA, AToM^3 , AGG, ATL [25], GreAT ...etc.
In our report we will present a simple comparison study between the most used tools, which are : AToM^3, AGG, ATL.

AToM^3 (A Tool for Multi-formalism and Meta-modeling)

- AToM^3 is a visual tool for modeling, meta-modeling implemented with python programming language on 2010 by J De Lara [27] at the university of Madrid. The main tasks in AToM^3 are meta-modeling,modeling and model transformation,
- For meta-modelling and modeling we can use ER, or UML Class diagram formalism,
- For model transformation AToM^3 is based on graph and graph transformation's rules,
- In AToM^3 we can define the execution order of rules,

- For each transformation's rule a condition is associated, AToM^3 use PAC (Positive Application Condition) systems to represents it, which means if the condition is verified, do the associated action.

AGG (Attributed Graph Grammar)

- AGG is a tool constructed at technical university of Berlin in 1997 by Gabriele Taentzer [28], implemented with java and based on graph transformation's approach, supported by a good graphical interface to design graphs and set the transformation rules,
- In agg we should define the execution order of these rules, because once we finish executing any rule we can not come back later,
- For each transformation rule a condition is associated, AGG use NAC (Negative Application Condition) systems to represents this condition, which means if the condition is not verified, do the associated action.

ATL (Atlas Transformation Language)

- ATL is a model transformation language and toolkit developed by the research team INIRIA (Jean Bezivin) [29], based on a hybrid transformation approach. In which the source and the target meta models are represented using XML format,
- ATL is supported by a set of development tools such as an editor, a compiler, a virtual machine, and a debugger,
- The condition of rule execution is expressed with ATL using OCL,
- We cannot define the execution's order of rules.

The comparison is resumed in Table 3.1.

Tools	Condition System	Rules execution order	Approach	Rules GUI
ATOM3	PAC(Python)	Yes	Declarative	Yes
AGG	NAC(Java)	Yes	Declarative	Yes
ATL	PAC(OCL)	No	Hybrid	No

Table 3.1: Comparaison between tranformation's tools.

In our project we choose to use ATL because it is a standard tool developed and supported by the OMG organization.

3.3 Definition of meta-models using EMF

3.3.1 EMF

EMF [33] is a free and open source tooling support within the Eclipse Framework, developed by the OMG on 2002, for specifying, constructing and managing meta-models through a graphical UI using Ecore, which is an EMF model used to define the main concepts of EMF like: EClass, EPackage, EOperation, EAttribute...etc.
After downloading and installing the Eclipse Modeling Framework from the link : `http://www.eclipse.org/modeling/emf/downloads/?`, we can edit meta-models with EMF by just creating a new *Ecore Model*.

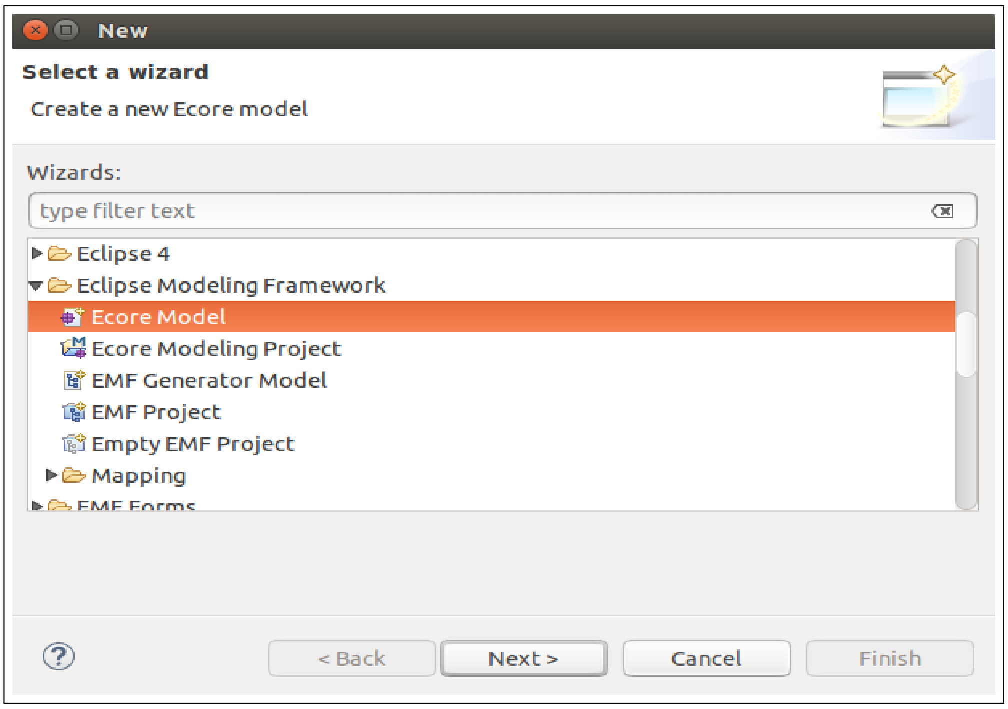

New
Select a wizard
Create a new Ecore model
Wizards:
type filter text
▶ Eclipse 4
▼ Eclipse Modeling Framework
Ecore Model
Ecore Modeling Project
EMF Generator Model
EMF Project
Empty EMF Project
▶ Mapping
▶ EMF Forms
< Back | Next > | Cancel | Finish

Figure 3.5: Ecore model.

Once creating an ecore model, we should initialize it with right click on it, and choosing *'Initialize Ecore Diagram...'*, after that we get a palette by which we can draw our meta model.

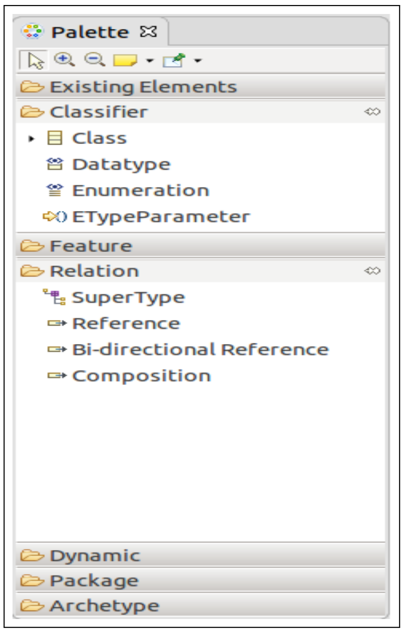


Figure 3.6: Palette.

3.3.2 Statecharts meta-model

- As shown in the Figure 3.7 a statechart diagram is composed by a set of nodes and a set of transitions,
- Each node can be the source or the target of a set of transitions,
- Each transition have one source and one target, and represented with an event and an action,
- A node can being an: initial state, finale state, state, fork and join node,
- For the state we have two type : simple and composite state, both of them has an entry and an exit internal transition or actions,
- A composite state can be itself composed of other states and all the elements of statechart diagram.

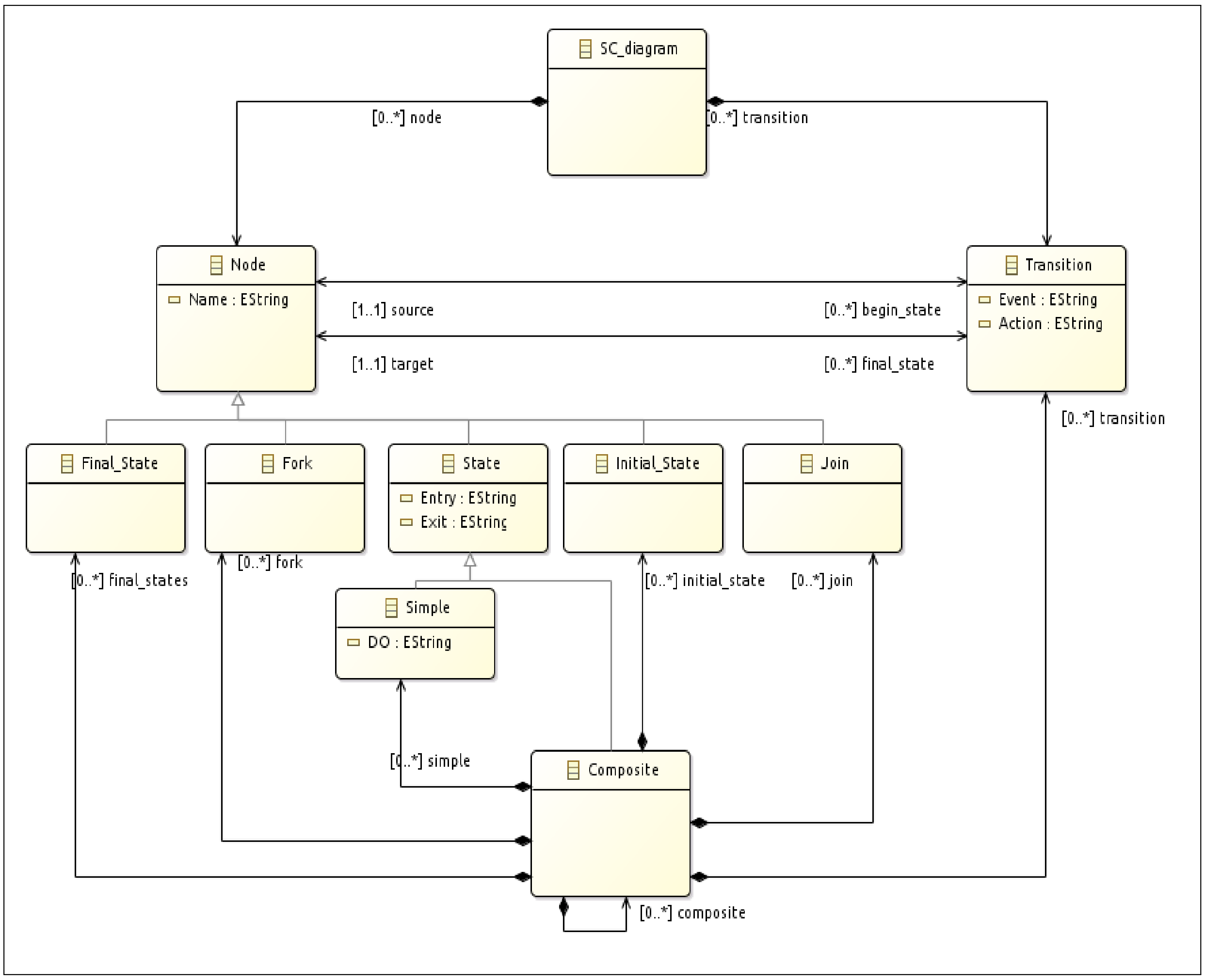


Figure 3.7: State diagram meta-model.

3.3.3 Petri Nets meta-model

- A petri nets is composed of a set of places and transitions,
- There is two types of arcs :
  - P2T arcs to link a place to transition,
  - T2P arcs to link a transition to place.

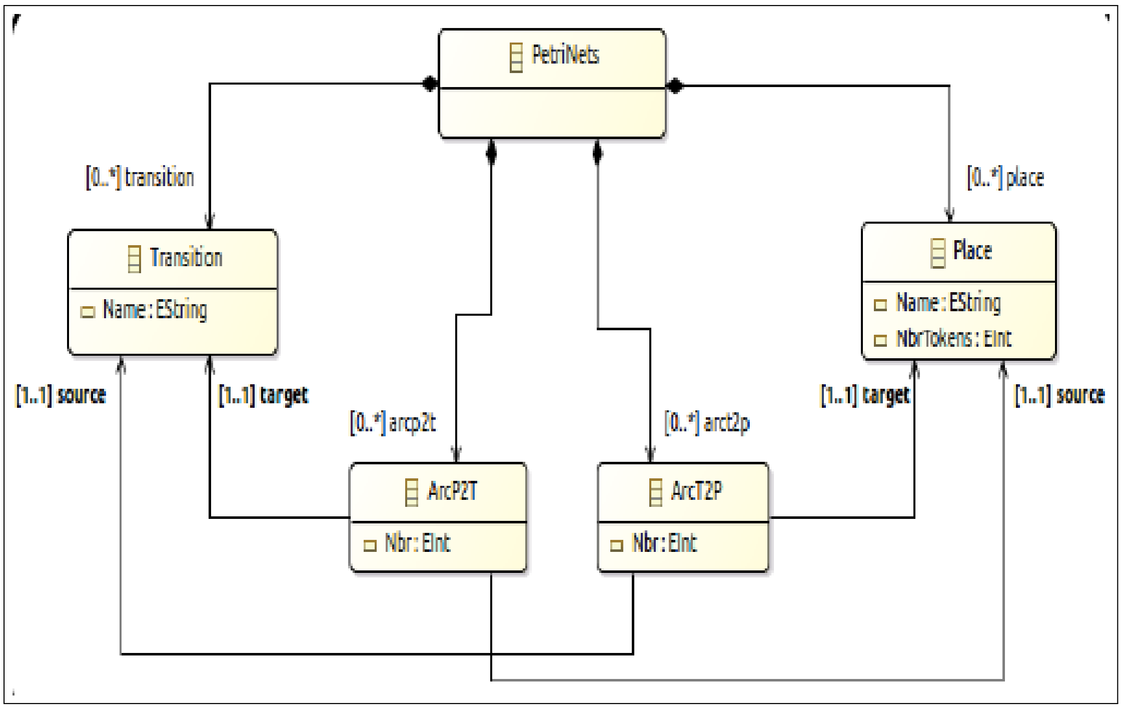
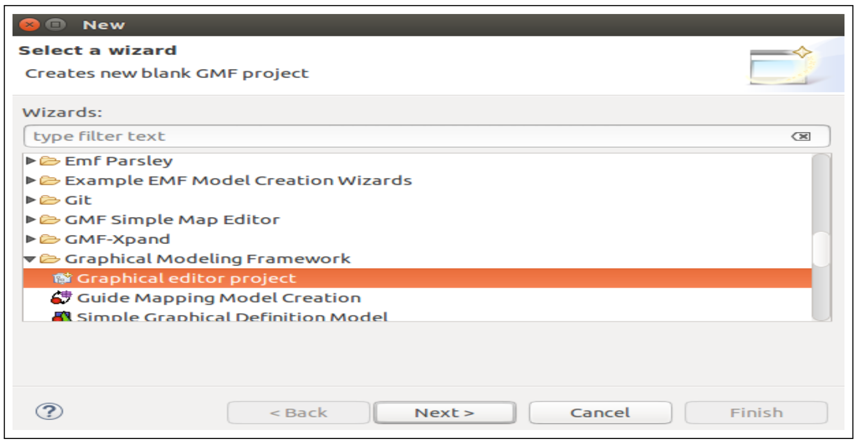


Figure 3.8: Petri nets meta-model.

3.4 Creation of statecharts and Petri nets editors using GMF

3.4.1 GMF

Graphical Modeling Framework [34] provides a set of components for developing and generating graphical editors based on EMF and GEF, it is used to define the figures, nodes, links...etc, that we will display on our diagram's editor.
After downloading and installing the Graphical Modeling Framework from the link : `https://www.eclipse.org/modeling/gmp/downloads/?project=gmf-tooling`, we can create step by step our editor by just creating a new *graphical editor project*.


New

Select a wizard

Creates new blank GMF project

Wizards:

type filter text

- Emf Parsley
- Example EMF Model Creation Wizards
- Git
- GMF Simple Map Editor
- GMF-Xpand
- Graphical Modeling Framework
  - Graphical editor project
  - Guide Mapping Model Creation
  - Simple Graphical Definition Model

Figure 3.9: Create GMF project.

Once creating it, the dashboard of GMF will be displayed as illustrated in Figure 3.10.

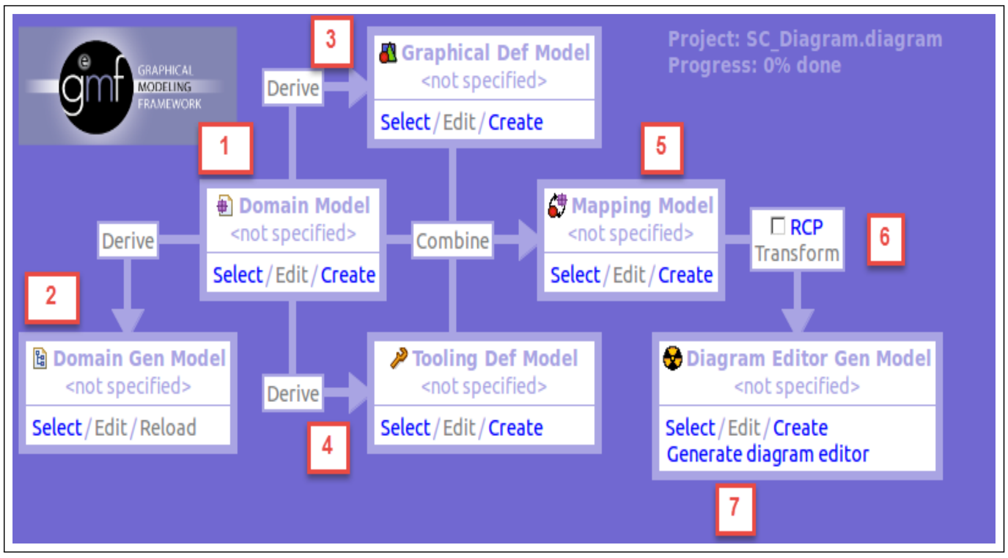


Figure 3.10: GMF dashboard.

It determines the set of steps that we should following to generate statecharts or Petri nets editors.

3.4.2 Creating statecharts diagram editor

Here we will execute the steps displayed on the dashboard with order.

- **Domain Model Definition**
  The top first step with GMF is to select, edit or create the meta-model that we want to used to create the graphical editor. For this meta-model, you have the choice between several kinds of format (Annotated Java code, Ecore model, Rose class model, UML model or XML Schema). In our case we are used the ecore meta-model of statechart.
  (Note: if this meta model is not yet created, we can do that by clicking on create on the domain model displayed on the dashboard).

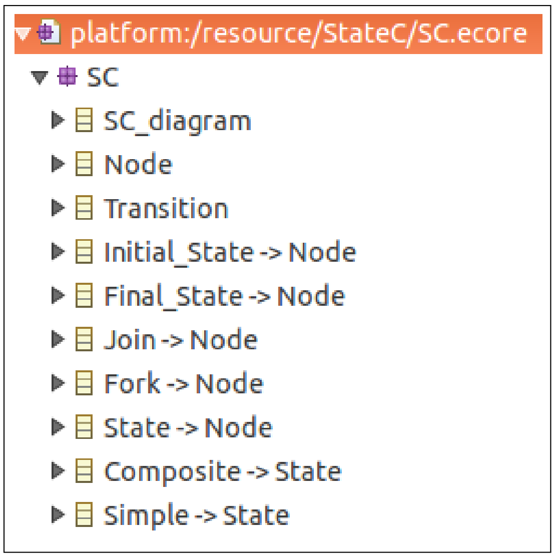

platform:/resource/StateC/SC.ecore
SC
SC_diagram
Node
Transition
Initial_State -> Node
Final_State -> Node
Join -> Node
Fork -> Node
State -> Node
Composite -> State
Simple -> State

Figure 3.11: SC domain model.

- **Domain Gen Model (.genmodel)**
  Generate SC.genmodel by just clicking on *'Derive'* in order to generate the domain model code with EMF. It's represented in the Figure 3.12.

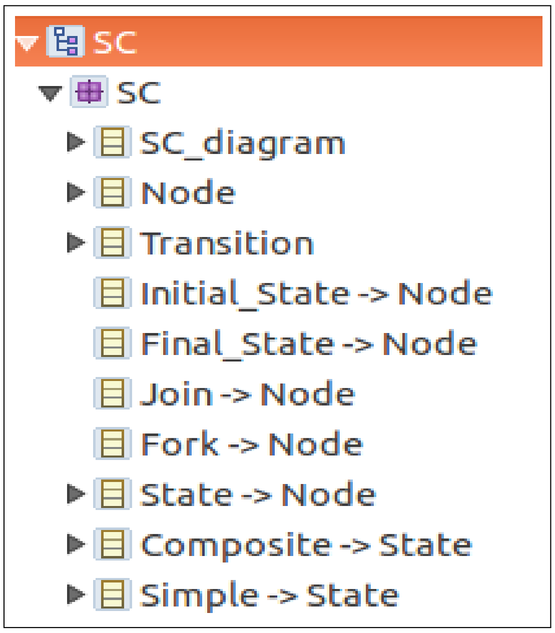
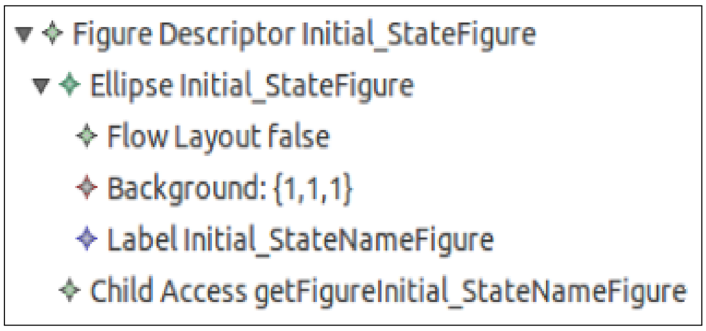

SC
SC
SC_diagram
Node
Transition
Initial_State -> Node
Final_State -> Node
Join -> Node
Fork -> Node
State -> Node
Composite -> State
Simple -> State

Figure 3.12: SC gen model.

- **Graphical Def Model (.gmfgraph)**
  Now it's time to define the graphical elements for the domain model, that means how to declare the graphical representation of the statechart elements like : initial state, final state, state, transition ...etc. We can do that by just clicking on the top derive button in GMF dashboard, and the graphical definition of

statechart elements will be presented like the following :

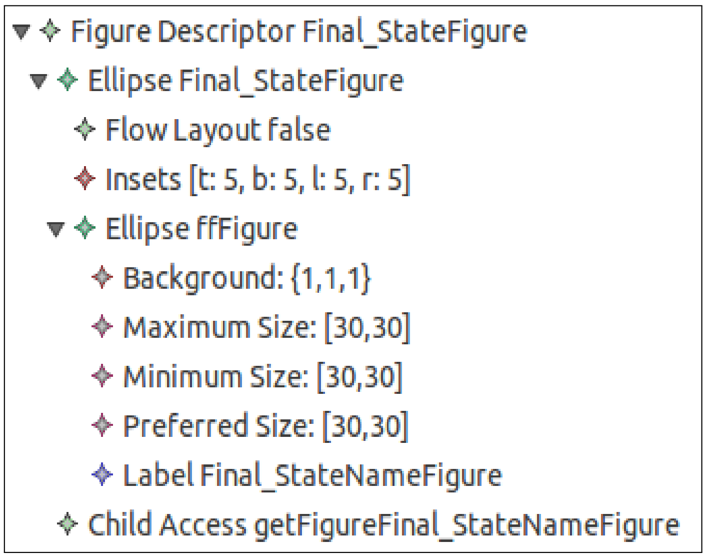
– **Initial state**



```
▼ ✧ Figure Descriptor Initial_StateFigure
  ▼ ✧ Ellipse Initial_StateFigure
      ✧ Flow Layout false
      ✧ Background: {1,1,1}
      ✧ Label Initial_StateNameFigure
    ✧ Child Access getFigureInitial_StateNameFigure
```

Figure 3.13: Initial state GMF graph.

– **Final state:** We represent the final state with the same manner then the initial state.



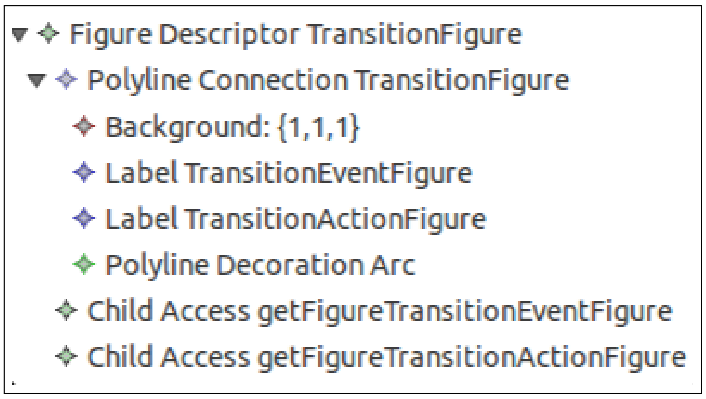
```
▼ ✧ Figure Descriptor Final_StateFigure
  ▼ ✧ Ellipse Final_StateFigure
      ✧ Flow Layout false
      ✧ Insets [t: 5, b: 5, l: 5, r: 5]
    ▼ ✧ Ellipse ffFigure
        ✧ Background: {1,1,1}
        ✧ Maximum Size: [30,30]
        ✧ Minimum Size: [30,30]
        ✧ Preferred Size: [30,30]
        ✧ Label Final_StateNameFigure
    ✧ Child Access getFigureFinal_StateNameFigure
```

Figure 3.14: Final state GMF graph.

Insets is used to make the margin between the external and the internel black circle.

– **Transition**

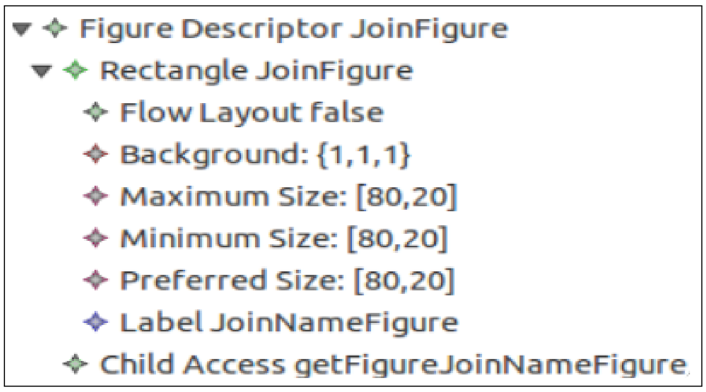
* A transition is a polyline that contain two labels, one for the event and the other for actions,
* We should define a child access for both of them.



```
▼ ✧ Figure Descriptor TransitionFigure
  ▼ ✧ Polyline Connection TransitionFigure
      ✧ Background: {1,1,1}
      ✧ Label TransitionEventFigure
      ✧ Label TransitionActionFigure
      ✧ Polyline Decoration Arc
    ✧ Child Access getFigureTransitionEventFigure
    ✧ Child Access getFigureTransitionActionFigure
```

Figure 3.15: Transition GMF graph.

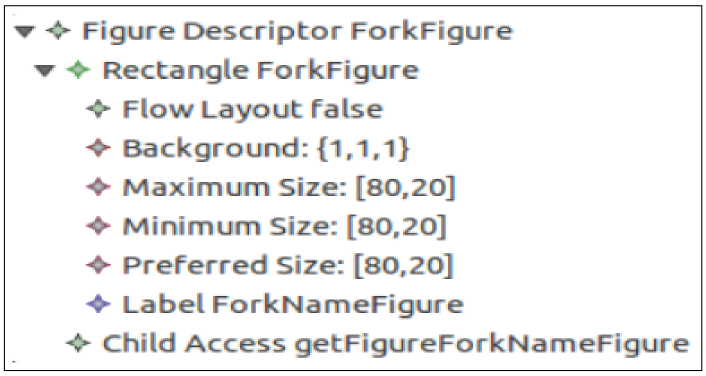
– **Join:** Join node is simply presented with a black rectangle.



```
▼ ✧ Figure Descriptor JoinFigure
  ▼ ✧ Rectangle JoinFigure
      ✧ Flow Layout false
      ✧ Background: {1,1,1}
      ✧ Maximum Size: [80,20]
      ✧ Minimum Size: [80,20]
      ✧ Preferred Size: [80,20]
      ✧ Label JoinNameFigure
    ✧ Child Access getFigureJoinNameFigure
```

Figure 3.16: Join GMF graph.

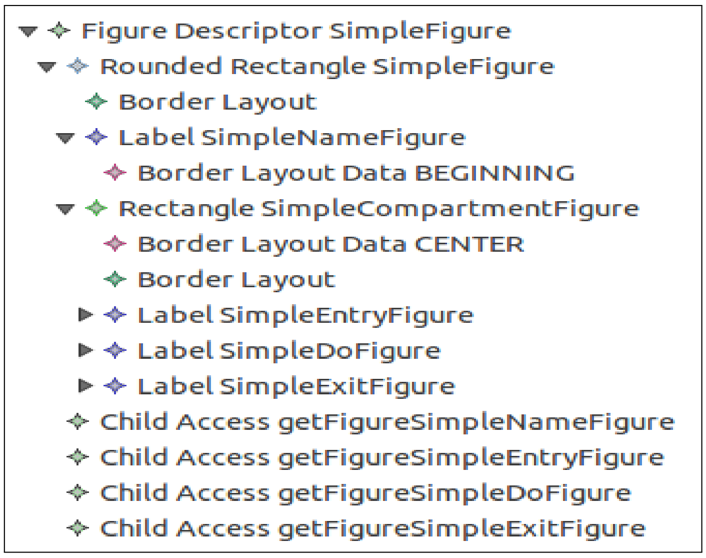
– **Fork:** like join, fork node also is presented with a black rectangle.



```
▼ ✧ Figure Descriptor ForkFigure
  ▼ ✧ Rectangle ForkFigure
      ✧ Flow Layout false
      ✧ Background: {1,1,1}
      ✧ Maximum Size: [80,20]
      ✧ Minimum Size: [80,20]
      ✧ Preferred Size: [80,20]
      ✧ Label ForkNameFigure
    ✧ Child Access getFigureForkNameFigure
```

Figure 3.17: Fork GMF graph.

– **Simple state**



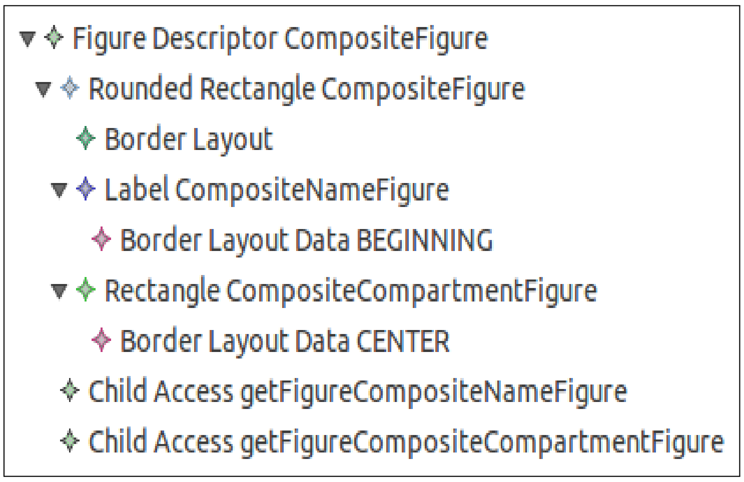
```
▼ ✧ Figure Descriptor SimpleFigure
  ▼ ✧ Rounded Rectangle SimpleFigure
      ✧ Border Layout
    ▼ ✧ Label SimpleNameFigure
        ✧ Border Layout Data BEGINNING
    ▼ ✧ Rectangle SimpleCompartmentFigure
        ✧ Border Layout Data CENTER
        ✧ Border Layout
      ▶ ✧ Label SimpleEntryFigure
      ▶ ✧ Label SimpleDoFigure
      ▶ ✧ Label SimpleExitFigure
    ✧ Child Access getFigureSimpleNameFigure
    ✧ Child Access getFigureSimpleEntryFigure
    ✧ Child Access getFigureSimpleDoFigure
    ✧ Child Access getFigureSimpleExitFigure
```

Figure 3.18: Simple state GMF graph.

∗ The state is represented by a *Rounded Rectangle*,

∗ We have defined a border layout in order to organize each composed elements.

– **Composite state:** With the same way we represent the composite state.



```
▼ ✧ Figure Descriptor CompositeFigure
  ▼ ✧ Rounded Rectangle CompositeFigure
      ✧ Border Layout
    ▼ ✧ Label CompositeNameFigure
        ✧ Border Layout Data BEGINNING
    ▼ ✧ Rectangle CompositeCompartmentFigure
        ✧ Border Layout Data CENTER
    ✧ Child Access getFigureCompositeNameFigure
    ✧ Child Access getFigureCompositeCompartmentFigure
```

Figure 3.19: Composite state GMF graph.

- **Tooling Def Model (.gmftool)**
  This file is used to define the palette of tools that we can use in the graphical editor. To generate it we just only need to click on the bottom derive button on

the dashboard.
The generated palette is displayed in the Figure 3.20.

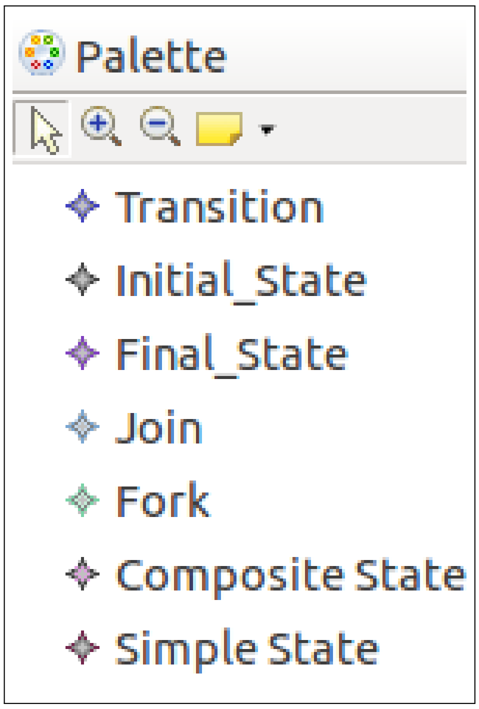

Palette
Transition
Initial_State
Final_State
Join
Fork
Composite State
Simple State

Figure 3.20: Palette graphic representation.

- **Mapping Model (.gmfmap)**
  This file links the domain model, the graphical model (.gmfgraph) and the tooling model (.gmftool) in order to map each element on ecore meta-model with its equivalent on the graphical and the tooling models (Figure 3.21).

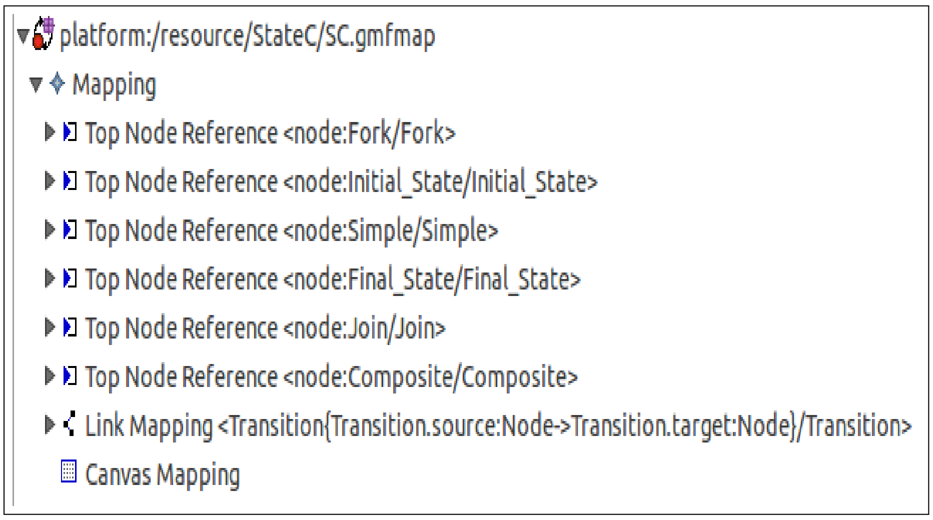

```
platform:/resource/StateC/SC.gmfmap
  Mapping
    Top Node Reference <node:Fork/Fork>
    Top Node Reference <node:Initial_State/Initial_State>
    Top Node Reference <node:Simple/Simple>
    Top Node Reference <node:Final_State/Final_State>
    Top Node Reference <node:Join/Join>
    Top Node Reference <node:Composite/Composite>
    Link Mapping <Transition{Transition.source:Node->Transition.target:Node}/Transition>
    Canvas Mapping
```

Figure 3.21: Statechart GMF map.

- **Diagram Editor Gen Model (.gmfgen)**
  This final file is used to generate the GMF graphical editor in addition to the EMF code generated by the *.genmodel* file, So we can generate two kind of graphical editor with GMF : a plug-in graphical editor integrated with Eclipse or as an RCP (Rich Client Platform) application which consists in an autonomous application. To generate a RCP application, just click into RCP on the dashboard. In our case we have created a GMF editor as an RCP, the result after running the diagram editor is presented in the Figure 3.22.

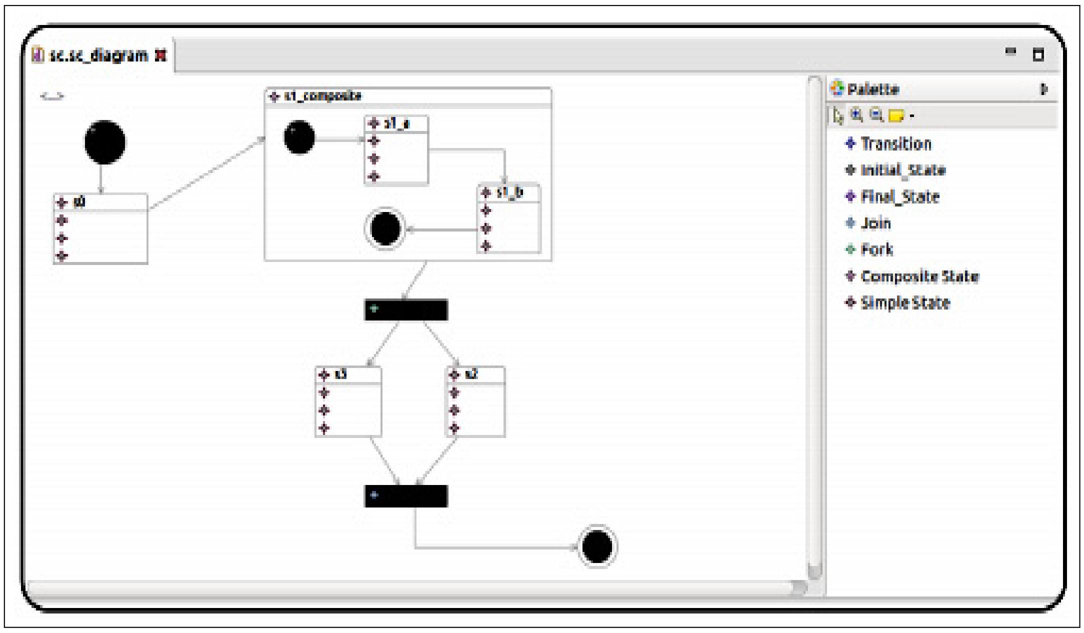


Figure 3.22: Statechart editor.

3.4.3 Creating Petri Nets diagram editor

Following the same steps we have created Petri nets editor, the result is in the Figure 3.23.

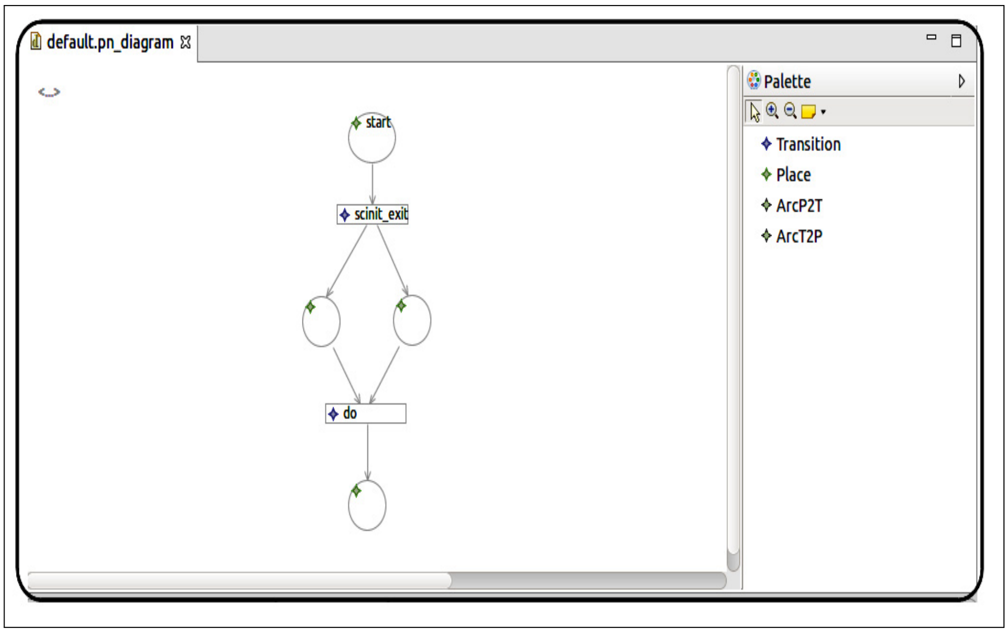


Figure 3.23: Petri nets editor

3.5 Conclusion

In the first part of this chapter we have explained the relation between OMG, MDE and MDA, until arriving to model transformation's concept. And we can resume this relation that the OMG, in its role as an industry driven organization that develops and maintains standards for developing complex distributed software systems, launched the Model Driven Architecture (MDA) as a framework of MDE standards. In the rest of the chapter we have explained with details the followed steps, starting by defining the source and the target meta-models using EMF, until creating diagrams editors using GMF.
What remain to us is to explain in the next chapter how the transformation's rules has been implemented using ATL .

Chapter 4

From Statecharts to Petri nets using ATL

4.1 Introduction

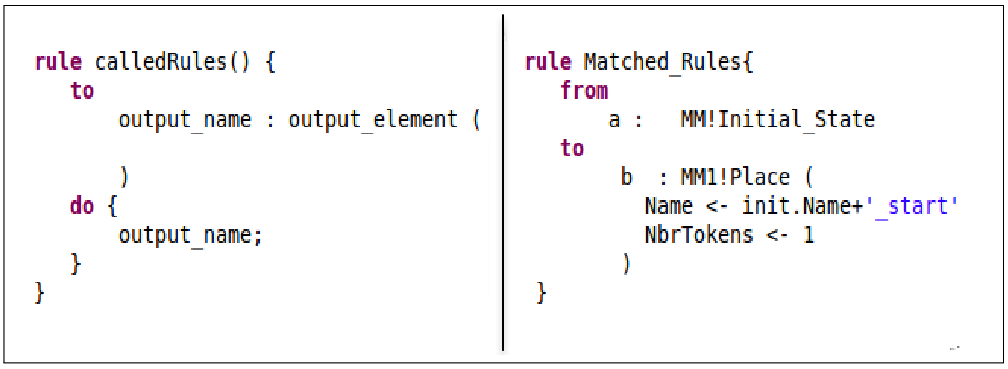
The ATL language offers ATL developers to design different kinds of ATL units. In the first part of this chapter we will try to explain briefly the main used concepts of ATL to define our transformation's grammars. Then we will present the set of used rules in order to transform any input statecharts diagrams composed from the following elements : simple state, composite state, initial state, final state, join and fork nodes to Petri nets model.

4.2 ATL units

There is a several types of ATL documents : module, query, library...etc.

- Module : An ATL module corresponds to a model to model transformation, its basic elements are:
  - An optional import section : enables to import some existing meta models or ATL libraries. In our grammar we have imported the meta models corresponds to the source and the target models like the following:
    – @path MM=/Rules/Meta-Models/SC.ecore
    – @path MM1=/Rules/Meta-Models/PN.ecore
  - Module declaration : to specify the source and the target meta models. In our transformation we have defined it like the following :
    module grammar;
    create OUT : MM1 **from** IN : MM;

- A set of helpers : can be viewed as an ATL equivalent to java methods, it will be used in our rules.
  helper def : name() : *OclAny* = OclUndefined;
- A set of rules : defines the way target models are generated from source ones. It can be matched rules (declarative programming) or called rules (imperative programming).



```
rule calledRules() {
   to
       output_name : output_element (

       )
   do {
       output_name;
   }
}
```

```
rule Matched_Rules{
   from
       a :   MM!Initial_State
   to
        b  : MM1!Place (
          Name <- init.Name+'_start'
          NbrTokens <- 1
        )
 }
```

Figure 4.1: Matched and called rules.

- Queries : A query is a type of transformation from a model to a primitive type, In the next chapter we will see that we have used an ATL query to transform the resulted petri nets model to textual representation of TINA tool, the core of the query is represented using OCL.
  query name = OclUndefined;

4.3 Rules implementation

What remain to us now is to explain the implemented rules using ATL, and for that we present two grammars of rules :
In the first grammar 7 rules has been defined to transform each node (simple state, join, fork ....) in the source model to its equivalent on the target Petri nets model.
To link between the resulted elements of the first grammar we have defined a second grammar in which it contains 13 rules and takes as source models the result of the first grammar and the source statechart diagram, and give as result the final Petri net target model.
All the rules are presented like the following:

Grammar 1:

- **Rule1**
  As the source and the target model are represented with XML, we should define a root's rule to express the structure of the XML target model file.

```
rule SC2PN{
     from
          sc : MM!SC_diagram
     to
          pn : MM1!PetriNets (
             place <- MM1!Place.allInstances(),
             transition <- MM1!Transition.allInstances(),
             arcp2t <- MM1!ArcP2T.allInstances(),
             arct2p <- MM1!ArcT2P.allInstances()
          )
}
```

- **Rule 2: Initial state node**
  Once the statechart diagram has been invoked the initial state will be activated, on Petri net it will be represented using a marked place and a transition.

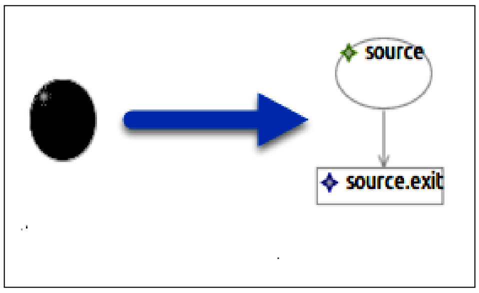


Figure 4.2: Mapping of initial state

```
rule initial_state2pn{
     from
          init :  MM!Initial_State
     to
          place : MM1!Place (
            Name <- init.Name,
            NbrTokens <- 1
          ),
          transition : MM1!Transition(
            Name <- init.Name+'_exit'
          ),
          arc : MM1!ArcP2T(
               Nbr <- 1,
               source <- place,
               target <- transition
          )
}
```

- **Rule 03: Final state node**
  On Petri nets it will be represented by creating a final place and to link it with an entry transition for each entry states.

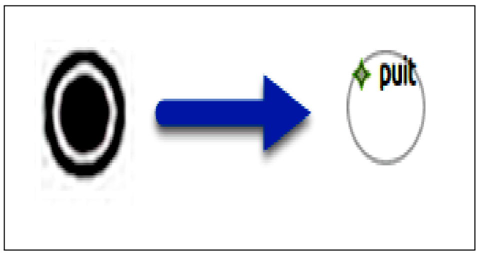


Figure 4.3: Mapping of final state

```
rule Final_state2PN{
     from
       final_state : MM!Final_State
     to
       place : MM1!Place(
         Name <- final_state.Name+'_puit'
     )
}
```

- **Rule 04: Simple state with empty *'Do'* activity:**
  Here we have three cases according to the transitions :
  - If the transition is immediately crossed.

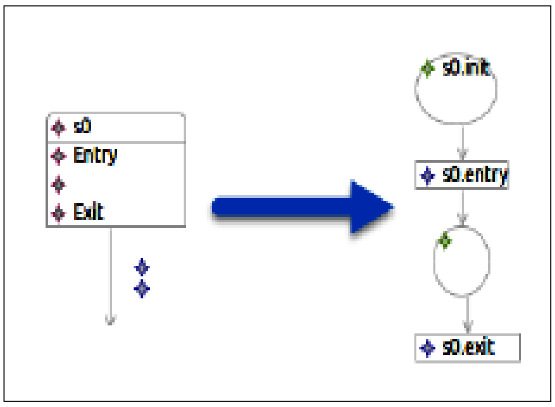


Figure 4.4: Mapping of simple state -*case 1*-

  - If the transition is crossed only after executing an action.

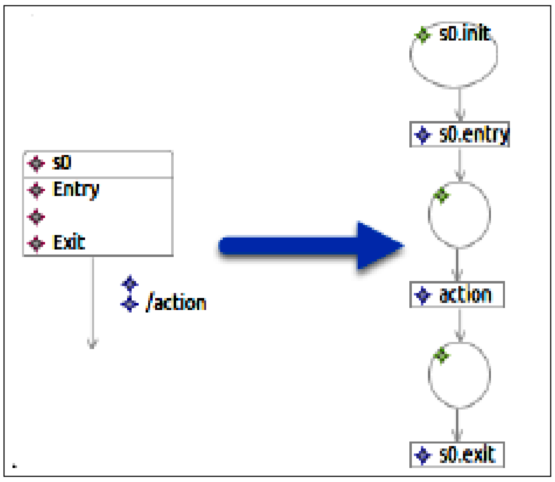


Figure 4.5: Mapping of simple state -*case 2*-

  - If the transition is crossed only after executing an action, and the action will never be executed without an incoming event.

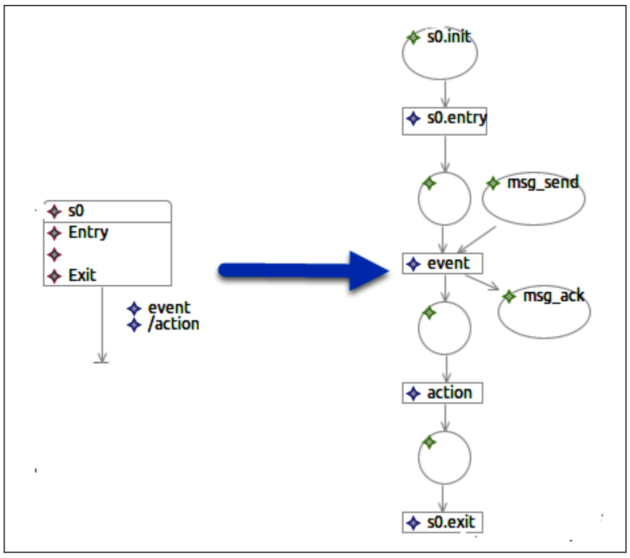


Figure 4.6: Mapping of simple state -*case 3*-

```
helper context MM!Simple def: Empty_StateActivity(): Boolean =
   if  self.DO=' ' then
        true
  else
       false
   endif;
   rule Simple2PN{
   from
    sc : MM!Simple(sc.Empty_StateActivity())
      using{
          transitions : Sequence(MM!Transition) = sc.begin_state;
        tr1 : Sequence(MM!Transition) = transitions->select(it |
            it.Event=' ' and it.Action=' ');
        tr2 : Sequence(MM!Transition) = transitions->select(it |
            it.Event=' ' and it.Action<>' ');
        tr3 : Sequence(MM!Transition) = transitions->select(it |
            it.Event<>' ' and it.Action<>' ');
      }
   to
     init_place: MM1!Place(
       Name <- sc.Name+ '_init'
     ),
     entry_transition: MM1!Transition(
       Name <- sc.Name+ '_entry'
     ),
     arc0: MM1!ArcP2T(
       source <- init_place,
        target <- entry_transition,
        Nbr <- 1
```

```
),
place_a0 :distinct MM1!Place foreach(t in tr1)(
  Name <- 'p'
),
arc_a0 :distinct MM1!ArcT2P foreach(t in tr1)(
   source <- entry_transition,
    target <- place_a0,
    Nbr <- 1
),
place_b0 : distinct MM1!Place foreach(t in tr2)(
   Name <- 'b'
),
arc_b0 :distinct MM1!ArcT2P foreach(t in tr2)(
   source <- entry_transition,
    target <- place_b0,
    Nbr <- 1
),
action_transition_b0:distinct MM1!Transition foreach(t in tr2)(
   Name <- 'action_'+ t.Action
),
arc_b1 :distinct MM1!ArcP2T foreach(t in tr2)(
   source <- place_b0,
    target <- action_transition_b0,
    Nbr <- 1
),
place_b1 :distinct MM1!Place foreach(t in tr2)(
   Name <- 'c'
),
arc_b2:distinct MM1!ArcT2P foreach(t in tr2)(
   source <- action_transition_b0,
    target <- place_b1,
    Nbr <- 1
),
arc_a1:distinct MM1!ArcP2T foreach(t in tr1)(
  Nbr <- 1,
  source <- place_a0,
  target <- exit_transition
),
arc_b3:distinct MM1!ArcP2T foreach(t in tr2)(
   source <- place_b1,
    target <- exit_transition,
    Nbr <- 1
),
place_c0 : distinct MM1!Place foreach(t in tr3)(
   Name <- 'h'
```

```
),
arc_c0 :distinct MM1!ArcT2P foreach(t in tr3)(
    source <- entry_transition,
     target <- place_c0,
     Nbr <- 1
),
event_transition:distinct MM1!Transition foreach(t in tr3)(
     Name <-'event_'+t.Event
),
arc_c1 :distinct MM1!ArcP2T foreach(t in tr3)(
   source <- place_c0,
    target <- event_transition,
     Nbr <- 1
),
place_msg :distinct MM1!Place foreach(t in tr3)(
    Name <- 'msg_send'
),
arc_msg:distinct MM1!ArcP2T foreach(t in tr3)(
    source <- place_msg,
     target <- event_transition,
     Nbr <- 1
),
place_ack :distinct MM1!Place foreach(t in tr3)(
    Name <- 'msg_ack'
),
arc_ack:distinct MM1!ArcT2P foreach(t in tr3)(
    source <- event_transition,
     target <- place_ack,
     Nbr <- 1
),
place_c1 :distinct MM1!Place foreach(t in tr3)(
    Name <- 'q'
),
arc_c2:distinct MM1!ArcT2P foreach(t in tr3)(
    source <- event_transition,
     target <- place_c1,
     Nbr <- 1
),
action_transition_c0:distinct MM1!Transition foreach(t in tr3)(
    Name <- 'action_'+ t.Action
),
arc_c3 :distinct MM1!ArcP2T foreach(t in tr3)(
    source <- place_c1,
     target <- action_transition_c0,
     Nbr <- 1
```

```
        ),
        place_c2 :distinct MM1!Place foreach(t in tr3)(
            Name <- 'z'
        ),
        arc_c4:distinct MM1!ArcT2P foreach(t in tr3)(
            source <- action_transition_c0,
             target <- place_c2,
             Nbr <- 1
        ),
        arc_c5:distinct MM1!ArcP2T foreach(t in tr3)(
            source <- place_c2,
             target <- exit_transition,
             Nbr <- 1
        ),
        exit_transition: MM1!Transition(
             Name <- sc.Name+ '_exit'
        )
}
```

- **Rule 05: Simple state with *'Do'* activity**
  Here also we have three cases according to the transitions :
  - If the transition is immediately crossed.

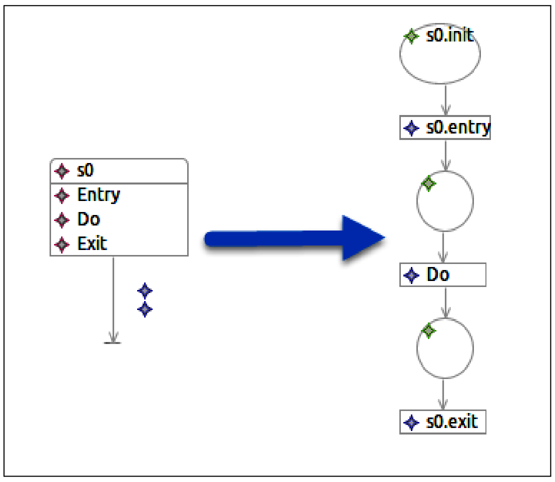


Figure 4.7: Mapping of simple state -*case 4*-

– If the transition is crossing only after executing an action.

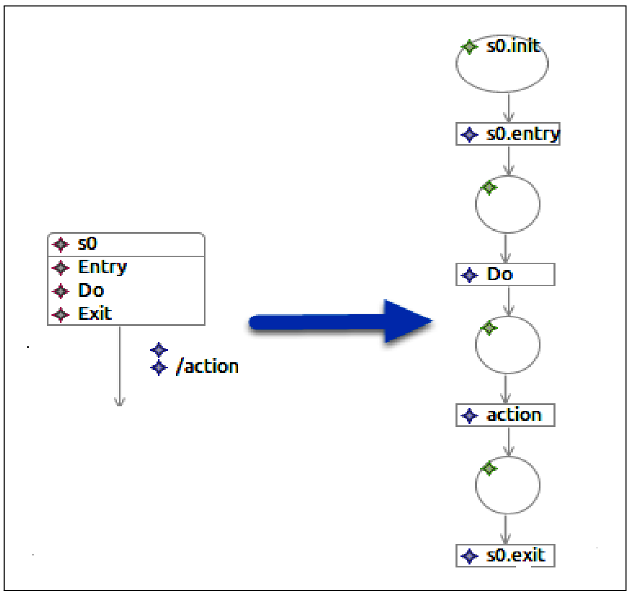


Figure 4.8: Mapping of simple state -*case* 5-

– If the transition is crossing only after executing an action, and the action will never be executed without an incoming event.

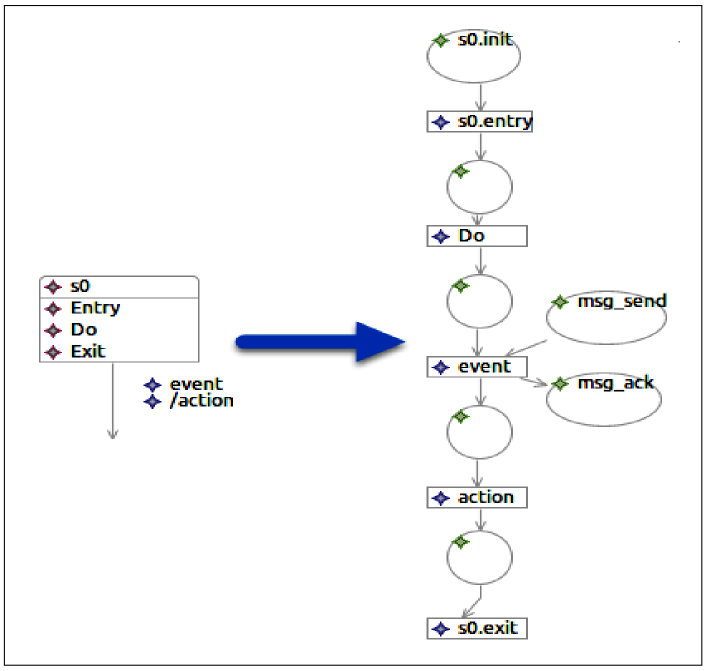


Figure 4.9: Mapping of simple state -*case* 6-

```
rule SimpleDo2PN{
   from
    sc : MM!Simple(not(sc.Empty_StateActivity()))
      using{
          transitions : Sequence(MM!Transition) = sc.begin_state;
        tr1 : Sequence(MM!Transition) = transitions->select(it |
            it.Event=' ' and it.Action='');
        tr2 : Sequence(MM!Transition) = transitions->select(it |
            it.Event=' ' and it.Action<>'');
        tr3 : Sequence(MM!Transition) = transitions->select(it |
            it.Event<>' ' and it.Action<>'');
      }
   to
     init_place: MM1!Place(
       Name <- sc.Name+ '_init'
     ),
     entry_transition: MM1!Transition(
       Name <- sc.Name+ '_entry'
     ),
     arc0: MM1!ArcP2T(
       source <- init_place,
        target <- entry_transition,
        Nbr <- 1
     ),
     place: MM1!Place(
       Name <- 'w'
     ),
     arc2 : MM1!ArcT2P(
       source <- entry_transition,
        target <- place,
        Nbr <- 1
     ),
     do_transition : MM1!Transition(
         Name <- 'Do_' + sc.DO
     ),
     arc3:MM1!ArcP2T(
       source <- place,
        target <- do_transition,
        Nbr <- 1
     ),
     place_a0 :distinct MM1!Place foreach(t in tr1)(
        Name <- 's'
     ),
     arc_a0 :distinct MM1!ArcT2P foreach(t in tr1)(
       Nbr <- 1,
   source <- do_transition,
   target <- place_a0
),
arc_a1 :distinct MM1!ArcP2T foreach(t in tr1)(
  Nbr <- 1,
   source <- place_a0,
   target <- exit_transition
),
place_bo :distinct MM1!Place foreach(t in tr2)(
    Name <- 'm'
),
arc_b0 :distinct MM1!ArcT2P foreach(t in tr2)(
  Nbr <- 1,
   source <- do_transition,
   target <- place_bo
),
action_transition :distinct MM1!Transition foreach(t in tr2 )(
       Name <- 'action_'+ t.Action
),
arc_b1 :distinct MM1!ArcP2T foreach(t in tr2)(
  Nbr <- 1,
   source <- place_bo,
   target <- action_transition
),
place_b1 :distinct MM1!Place foreach(t in tr2)(
    Name <- 'f'
),
arc_b2 :distinct MM1!ArcT2P foreach(t in tr2)(
  Nbr <- 1,
   source <- action_transition,
   target <- place_b1
),
arc_b3 : distinct MM1!ArcP2T foreach(t in tr2)(
  Nbr <- 1,
   source <- place_b1,
   target <- exit_transition
),
place_c0 :distinct MM1!Place foreach(t in tr3)(
    Name <- 'o'
),
arc_c0 :distinct MM1!ArcT2P foreach(t in tr3)(
  Nbr <- 1,
   source <- do_transition,
   target <- place_c0
),
event_transition:distinct MM1!Transition foreach(t in tr3)(
     Name <-'event_'+t.Event
),
arc_c1 :distinct MM1!ArcP2T foreach(t in tr3)(
   source <- place_c0,
    target <- event_transition,
     Nbr <- 1
),
place_msg :distinct MM1!Place foreach(t in tr3)(
    Name <- 'msg_send'
),
arc_msg:distinct MM1!ArcP2T foreach(t in tr3)(
    source <- place_msg,
     target <- event_transition,
     Nbr <- 1
),
place_ack :distinct MM1!Place foreach(t in tr3)(
    Name <- 'msg_ack'
),
arc_ack:distinct MM1!ArcT2P foreach(t in tr3)(
    source <- event_transition,
     target <- place_ack,
     Nbr <- 1
),
place_c1 :distinct MM1!Place foreach(t in tr3)(
    Name <- 'i'
),
arc_c2:distinct MM1!ArcT2P foreach(t in tr3)(
    source <- event_transition,
     target <- place_c1,
     Nbr <- 1
),
action_transition_c0:distinct MM1!Transition foreach(t in tr3)(
    Name <- 'action_'+ t.Action
),
arc_c3 :distinct MM1!ArcP2T foreach(t in tr3)(
    source <- place_c1,
     target <- action_transition_c0,
     Nbr <- 1
),
place_c2 :distinct MM1!Place foreach(t in tr3)(
    Name <- 'u'
),
arc_c4:distinct MM1!ArcT2P foreach(t in tr3)(
    source <- action_transition_c0,
             target <- place_c2,
             Nbr <- 1
        ),
        arc_c5:distinct MM1!ArcP2T foreach(t in tr3)(
            source <- place_c2,
             target <- exit_transition,
             Nbr <- 1
        ),
        exit_transition: MM1!Transition(
             Name <- sc.Name+ '_exit'
        )
   }
```

- **Rule 06: Join node**
  A join node it's a control element to synchronise a multiple input streams. To transform it to petri nets we just need to create a transition with the same synchronisation properties.

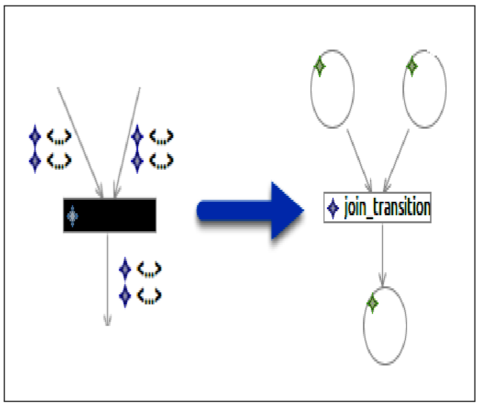


Figure 4.10: Mapping of Join node

```
rule Join2PN{
  from
    join_noeud : MM!Join
    using{
       input_transitions : Sequence(MM!Transition) = join_noeud.final_state;
    }
   to
    pl :distinct MM1!Place foreach(tr in input_transitions)(
       Name <- tr.source.Name+'_t'
    ),
    arc :distinct MM1!ArcP2T foreach(tr in input_transitions)(
        source <- pl,
        target <- transition,
            Nbr <-1
        ),
        transition : MM1!Transition(
            Name <- join_noeud.Name+'_join'
        )
}
```

- **Rule 07: Fork node**
  A fork node it's a control node to duplicate an input stream into several output streams. To transform it to petri nets we just need to create a transition with the same synchronisation properties.

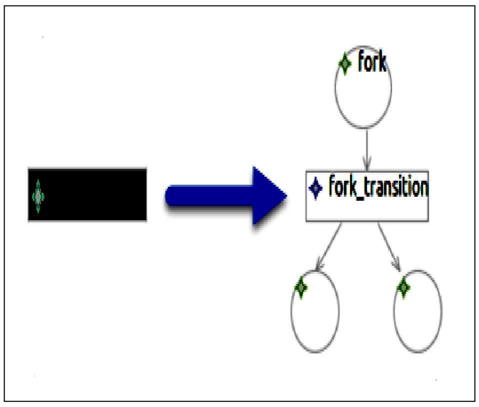


Figure 4.11: Mapping of fork node.

```
rule Fork2PN{
     from
       fork_node : MM!Fork
     to
       place : MM1!Place(
          Name <- 'fork'
       ),
       arc : MM1!ArcP2T(
         source <- place,
          target <- transition0,
          Nbr <- 1
       ),
       transition0 : MM1!Transition(
         Name <- 'fork_transition'
       )
 }
```

Grammar 2:

The Grammar 02 contain 13 rules grouped into two parts. In the first part we have maked the set of rules to copy all the places, transitions, and arcs resulted from the first grammar into the target model of the second grammar. And the second part represents the rules to link between these elements in order to get finally the complete target model.

- **Rule 01**

  The root's rule of the second grammar is represented like the following:

```
     rule PN2PN{
    from
      pn1 : MM1!PetriNets,
      sc : MM!SC_diagram
    to
      pn2 : MM2!PetriNets(
         place <- pn1.place,
         transition <- pn1.transition,
         arcp2t <- pn1.arcp2t,
         arct2p <- pn1.arct2p,
         place <- MM2!Place.allInstances(),
         transition <- MM2!Transition.allInstances(),
         arct2p <- MM2!ArcT2P.allInstances(),
         arcp2t <- MM2!ArcP2T.allInstances()
      )
}
```

- **Rule 02**

  Copy all the places of the target model of the grammar 1 to the target model of the grammar2.

```
rule Place2Place{
    from
      pl1 : MM1!Place
    to
      pl2 : MM2!Place(
        Name <- pl1.Name,
         NbrTokens <- pl1.NbrTokens)}
```

- **Rule 03**
  Copy all the transitions of the target model of the grammar 1 to the target model of the grammar2.

```
rule Transition2Transition{
     from
       tr1 : MM1!Transition
     to
       tr2 : MM2!Transition(
         Name <- tr1.Name)
         }
```

- **Rule 04**
  Copy all the arcs with type 'place to transition' of the target model of the grammar 1 to the target model of the grammar2.

```
rule ArcPT2ArcPT{
     from
       arcp2t1 : MM1!ArcP2T
     to
       arct2p2 : MM2!ArcP2T(
         source <- arcp2t1.source,
          target <- arcp2t1.target,
          Nbr <- arcp2t1.Nbr
       )
}
```

- **Rule 05**
  Copy all the arcs with type 'transition to place' of the target model of the grammar 1 to the target model of the grammar2.

```
rule ArcTP2ArcTP{
     from
       arct2p1 : MM1!ArcT2P
     to
       arct2p2 : MM2!ArcT2P(
         source <- arct2p1.source,
          target <- arct2p1.target,
          Nbr <- arct2p1.Nbr
       )
}
```

- **Rule 06**
  For each transition that have as source an Initial state and as target a state **s**, create an arc from the *exit-transition* corresponds to initial state to the *initial-place* corresponds to the state ***s***.

```
helper def : places : Sequence(MM2!Place) = MM2!Place->allInstances();
helper def : transitions : Sequence(MM1!Transition) =
    MM1!Transition->allInstances();
helper def : get_exit_Transition(name:String) : MM2!Transition =
    thisModule.transitions->select(it | it.Name = name+'_exit')->first();
helper def : get_init_Place(name:String) : MM2!Place =
    thisModule.places->select(it | it.Name = name+'_init')->first();

rule from_init{
    from
     tr: MM!Transition((tr.source.oclIsTypeOf(MM!Initial_State))
     .and((tr.target.oclIsTypeOf(MM!Simple))
     .or(tr.target.oclIsTypeOf(MM!Composite))))
     to
       arc2 : MM2!ArcT2P(
         source <- thisModule.get_exit_Transition(tr.source.Name+'init'),
          target <- thisModule.get_init_Place(tr.target.Name),
          Nbr <- 1
       )
}
```

- **Rule 07**
  For each transition that have as source a state **s** and as target a finale state, create an arc from the *exit-transition* of the state **s** to the final place.

```
helper def : get_final_place(name:String) : MM1!Place =
    thisModule.places->select(it | it.Name = name+'_puit')->first();

rule to_final{
    from
         tr: MM!Transition (((tr.source.oclIsTypeOf(MM!Simple))
         .or(tr.source.oclIsTypeOf(MM!Composite)))
         .and(tr.target.oclIsTypeOf(MM!Final_State)))
     to   arc2 : MM2!ArcT2P(
         source <- thisModule.get_exit_Transition(tr.source.Name),
          target <- thisModule.get_final_place(tr.target.Name),
          Nbr <- 1)}
```

- **Rule 08**
  For each transition when the source and the target are states (simple or composite state), create an arc from the *exit-transition* corresponds to the source node, to the *init-place* corresponds to the target node.

```
rule from_state_to_state{
    from
       tr: MM!Transition (((tr.source.oclIsTypeOf(MM!Simple))
       .or(tr.source.oclIsTypeOf(MM!Composite)))
       .and((tr.target.oclIsTypeOf(MM!Simple))
       .or(tr.target.oclIsTypeOf(MM!Composite))))
     to
       arc2 :MM2!ArcT2P(
         source <- thisModule.get_exit_Transition(tr.source.Name),
          target <- thisModule.get_init_Place(tr.target.Name),
          Nbr <- 1
      )
      }
```

- **Rule 09**
  Link each fork node with its incoming node.

```
helper def : get_Fork_place(name:String) : MM1!Place =
    thisModule.places->select(it | it.Name = name+'_f')->first();

 rule to_fork{
     from
       tr: MM!Transition (((tr.source.oclIsTypeOf(MM!Simple))
       .or(tr.source.oclIsTypeOf(MM!Composite)))
       .and(tr.target.oclIsTypeOf(MM!Fork)))
     to
       arc : MM2!ArcT2P(
        source <- thisModule.get_exit_Transition(tr.source.Name),
        target <- thisModule.get_Fork_place(tr.source.Name),
        Nbr <- 1
       )
}
```

- **Rule 10**
  Link each fork node with all its outgoing nodes.

```
 helper def : get_Fork_Transition(name:String) : MM1!Transition =
     thisModule.transitions->select(it | it.Name = name+'_fork')->first();

   rule from_fork{
     from
        tr: MM!Transition ((tr.source.oclIsTypeOf(MM!Fork))
        .and((tr.target.oclIsTypeOf(MM!Simple))
        .or(tr.target.oclIsTypeOf(MM!Composite))))
     to
        arc : MM2!ArcT2P(
         source <- thisModule.get_Fork_Transition(tr.source.Name),
          target <- thisModule.get_init_Place(tr.target.Name),
          Nbr <- 1
        )
}
```

- **Rule 11**
  Link each join node with all its incoming nodes.

```
helper def : get_t_place(name:String) : MM2!Place =
    thisModule.places->select(it | it.Name = name+'_t')->first();

rule to_join{
    from
       tr: MM!Transition (((tr.source.oclIsTypeOf(MM!Simple))
       .or(tr.source.oclIsTypeOf(MM!Composite)))
       .and(tr.target.oclIsTypeOf(MM!Join)))
     to
       arc0 : MM2!ArcT2P(
           source <- thisModule.get_exit_Transition(tr.source.Name),
            target <- thisModule.get_t_place(tr.source.Name),
            Nbr <-1
       )
}
```

- **Rule 12**
  Link each join node with its outgoing node.

```
rule from_join{
    from
       tr: MM!Transition ((tr.source.oclIsTypeOf(MM!Join))
        .and((tr.target.oclIsTypeOf(MM!Simple))
        .or(tr.target.oclIsTypeOf(MM!Composite))))
      to
        arc0 : MM2!ArcT2P(
            source <- thisModule.get_Join_Transition(tr.source.Name),
             target <- thisModule.get_init_Place(tr.target.Name),
             Nbr <-1
        )
}
```

- **Rule 13: Composite state**
  -A state in which it's composed from a several others state.
  -To transform it we just need to apply the previous rules for the simple state, initial state, final state, fork node and join node rules. and Composite2PN rule in which it's represented like the following:

```
rule Composite2PN{
     from
      sc : MM!Composite
        using{
          trans : Sequence(MM!Transition) = sc.final_state;
            transitions : Sequence(MM!Transition) = sc.begin_state;
          tr1 : Sequence(MM!Transition) = transitions->select(it | it.Event=' '
              and it.Action=' ');
          tr2 : Sequence(MM!Transition) = transitions->select(it | it.Event=' '
              and it.Action<>' ');
          tr3 : Sequence(MM!Transition) = transitions->select(it | it.Event<>' '
              and it.Action<>' ');
        }
     to
       init_place: MM2!Place(
         Name <- sc.Name+ '_init'
       ),
       entry_transition: MM2!Transition(
         Name <- sc.Name+ '_entry'
       ),
       arc_01 :distinct MM2!ArcT2P foreach(t in trans)(
           source <- thisModule.get_exit_Transition(t.source.Name),
            target <-init_place,
            Nbr <- 1
       ),
       arc0: MM2!ArcP2T(
         source <- init_place,
   target <- entry_transition,
   Nbr <- 2
),
arc1: MM2!ArcT2P(
  source <- entry_transition,
   target <-
       thisModule.getCompsite_initial_state(sc.initial_state.first().Name),
   Nbr <- 1
),
tra : MM2!Transition(
   Name <- 'q '
),
arc2:MM2!ArcP2T(
    source <- thisModule.get_final_place(sc.final_states.first().Name),
    target <- tra,
    Nbr <- 1
),
arc_00 :distinct MM2!ArcT2P foreach(t in transitions)(
    source <- exit_transition,
     target <-thisModule.get_init_Place(t.target.Name),
     Nbr <- 1
),
place_a0 :distinct MM2!Place foreach(t in tr1)(
   Name <- 'p'
),
arc_a0 :distinct MM2!ArcT2P foreach(t in tr1)(
    source <- tra,
     target <- place_a0,
     Nbr <- 1
),
place_b0 : distinct MM2!Place foreach(t in tr2)(
    Name <- 'b'
),
arc_b0 :distinct MM2!ArcT2P foreach(t in tr2)(
    source <- tra,
     target <- place_b0,
     Nbr <- 1
),
action_transition_b0:distinct MM2!Transition foreach(t in tr2)(
    Name <- 'action_'+ t.Action
),
arc_b1 :distinct MM2!ArcP2T foreach(t in tr2)(
    source <- place_b0,
     target <- action_transition_b0,
     Nbr <- 1
```

```
),
place_b1 :distinct MM2!Place foreach(t in tr2)(
    Name <- 'c'
),
arc_b2:distinct MM2!ArcT2P foreach(t in tr2)(
    source <- action_transition_b0,
     target <- place_b1,
     Nbr <- 1
),
arc_a1:distinct MM2!ArcP2T foreach(t in tr1)(
  Nbr <- 1,
   source <- place_a0,
   target <- exit_transition
),
arc_b3:distinct MM2!ArcP2T foreach(t in tr2)(
    source <- place_b1,
     target <- exit_transition,
     Nbr <- 1
),
place_c0 : distinct MM2!Place foreach(t in tr3)(
    Name <- 'h'
),
arc_c0 :distinct MM2!ArcT2P foreach(t in tr3)(
    source <- tra,
     target <- place_c0,
     Nbr <- 1
),
event_transition:distinct MM2!Transition foreach(t in tr3)(
     Name <-'event_'+t.Event
),
arc_c1 :distinct MM2!ArcP2T foreach(t in tr3)(
   source <- place_c0,
    target <- event_transition,
     Nbr <- 1
),
place_msg :distinct MM2!Place foreach(t in tr3)(
    Name <- 'msg_send'
),
arc_msg:distinct MM2!ArcP2T foreach(t in tr3)(
    source <- place_msg,
     target <- event_transition,
     Nbr <- 1
),
place_ack :distinct MM2!Place foreach(t in tr3)(
    Name <- 'msg_ack'
```

```
        ),
        arc_ack:distinct MM2!ArcT2P foreach(t in tr3)(
            source <- event_transition,
             target <- place_ack,
             Nbr <- 1
        ),
        place_c1 :distinct MM2!Place foreach(t in tr3)(
            Name <- 'q'
        ),
        arc_c2:distinct MM2!ArcT2P foreach(t in tr3)(
            source <- event_transition,
             target <- place_c1,
             Nbr <- 1
        ),
        action_transition_c0:distinct MM2!Transition foreach(t in tr3)(
            Name <- 'action_'+ t.Action
        ),
        arc_c3 :distinct MM2!ArcP2T foreach(t in tr3)(
            source <- place_c1,
             target <- action_transition_c0,
             Nbr <- 1
        ),
        place_c2 :distinct MM2!Place foreach(t in tr3)(
            Name <- 'z'
        ),
        arc_c4:distinct MM2!ArcT2P foreach(t in tr3)(
            source <- action_transition_c0,
             target <- place_c2,
             Nbr <- 1
        ),
        arc_c5:distinct MM2!ArcP2T foreach(t in tr3)(
            source <- place_c2,
             target <- exit_transition,
             Nbr <- 1
        ),
        exit_transition: MM2!Transition(
             Name <- sc.Name+ '_exit'
        )
}
```

4.4 Conclusion

What has been presented in this chapter is two grammars of rules in which they are organized like the follwoing :

- Grammar 1 : 7 rules.
  - Statechart to Petri net root rule,
  - Initial state to Petri net,
  - Final state to Petri net,
  - Simple state with do activity to Petri net,
  - Simple state without do activity to Petri net,
  - Fork node to Petri net,
  - Join node to Petri net.
- Grammar 2 : 13 rules.
  - 5 rules to copying the resulted elements of the first grammar to the target model of the second grammar,
  - Composite state to Petri net,
  - The other rules to link the resulted Petri net elements.

In the next chapter we will present the formal verification techniques and tools, with the goal to verify the Petri net model resulted from the transformation.

Chapter 5

Formal model checking

5.1 Introduction

In chapter 2, the semi formalization of Petri nets has been presented, and we have said that a Petri nets it's a model in which it's composed from a set of places, transitions and arcs, furthermore we have briefly explained the concept of marking and firing transitions. After transforming the statechart source model to formal Petri net, we are now in the core of formal verification, and we will try in this chapter to explain the main formal concepts used to verify and analyse Petri nets in order to complete the final step of our proposed approach.

5.2 Analysis Of Petri nets

The study and the analysis of any system with Petri net is realized following these three steps:

- Represent the bahvior of system with a Petri net model,
- Specify the set of properties that you want to verify with a logical specification,
- Analyze the system and verify if it satisfies the input properties,

This process of verification is a very important approach to obtain a good evaluation of system.

5.2.1 Petri nets properties

One thing that makes Petri nets interesting is that they provide a balance between modeling power and analyzability. It has several type of mathematical properties, in our approach we will try to verify only four ones, which are : reachablity, livennes, boundeness, reversibility [12].

- Reachability: Reachability is a fundamental basis for studying the dynamic properties of any system. The firing of an enabled transition will change the token distribution (marking) in a net. A sequence of firings will result in a sequence of markings. A marking M, is said to be reachable from a marking M_0 if there exists a sequence of firings that transforms M_0 to M,
- Boundedness: A Petri net (N,M_0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from the initial marking M_0,
- Liveness: The concept of liveness is closely related to the complete absence of deadlocks in operating systems. A Petri net (N, M_0) is said to be live (or equivalently M_0 is said to be a live marking for N) if, no matter what marking has been reached from M_0, it is possible to ultimately fire any transition of the net by progressing through some further firing sequence,
- Reversibility: A Petri net (N,M_0) is said to be reversible if, for each marking M in R(M_0), M_0 is reachable from M.

5.2.2 Analysis methods

The methods of analysis of Petri nets may be classified into the following three groups [12]:

- Coverability (reachability) tree method,
- Algebraic approach,
- Reduction or decomposition techniques.

The first method involves essentially the enumeration of all reachable markings or their coverable markings. It should be able to apply to all classes of nets, but is limited to small nets due to the complexity of the state-space explosion. On the other hand, algebraic and reduction techniques are powerful but in many cases they are applicable only to special subclasses of Petri nets or special situations [12].

Coverability Tree

The coverability tree for a Petri net (N, M_0) is constructed by the following algorithm.

- Step 1) Label the initial marking M_0 as the root and tag it *'new'*
- Step 2) While *'new'* markings exist, do the following:
  - Step 2.1) Select a new marking M.
  - Step 2.2) If M is identical to a marking on the path from the root to M, then tag M *"old"* and go to another new marking.
  - Step 2.3) If no transitions are enabled at M, tag M *'dead-end'*.
  - Step 2.4) While there exist enabled transitions at M, do the following for each enabled transition tag M:
    - Step 2.4.1) Obtain the marking M that results from firing *t* at M.
    - Step 2.4.2) On the path from the root to M if there exists a marking M" such that M'(p) $\geq$ M"(p) for each place p and M' $\neq$ M", i.e., M is coverable, then replace M"(p) by ω for each p such that M'(p) $>$ M"(p).
    - Step 2.4.3) Introduce M' as a node, draw an arc with label *t* from M to M', and tag M' *'new'*.

Some of the properties that can be studied by using the coverability tree T for a Petri net (N, M_0) are the following:

- A net (N, M_0) is bounded and thus R(M_0) is finite iff (if and only if) ω does not appear in any node labels in T.
- A transition *t* is dead iff it does not appear as an arc label in T.
- If M is reachable from M_0 or then there exists a node labeled M such that M $\leq$ M.

Example: The coverability tree of the petri net illustrated in Figure 5.1 is represented in the Figure 5.2.

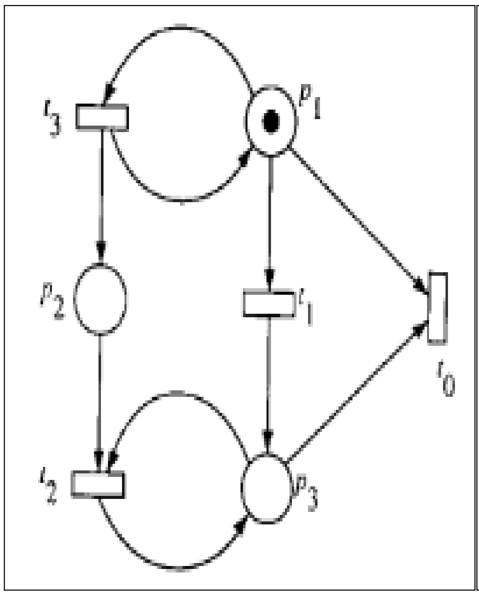


Figure 5.1: Petri net example.

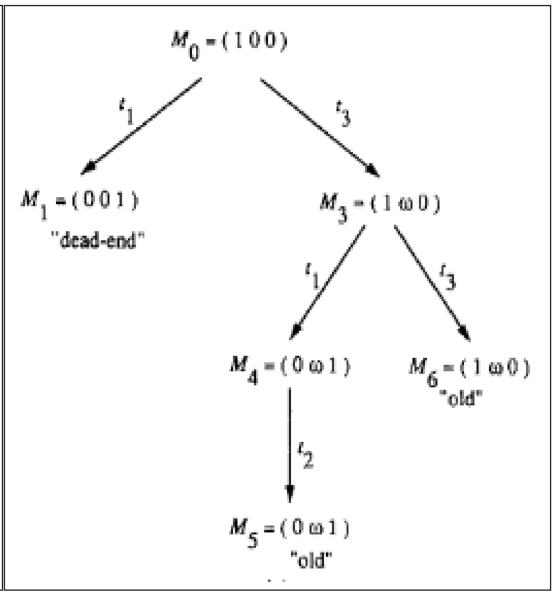


Figure 5.2: Coverability tree.

Algebraic methods

This idea makes it possible to study the properties of a Petri net irrespective of the initial marking. A net is structurally bounded if it is bounded for any finite initial marking, and if for Any initial marking the network is alive, we will deduce that the network is structurally living.

Simple reduction rules for analysis

To facilitate the analysis of a large system, we often reduce the system model to a simpler one, while preserving the system properties to be analyzed. Conversely, techniques to transform an abstracted model into a more refined model in a hierarchical manner can be used for synthesis. There exist many transformation techniques for Petri nets. In this section, we present only the simples transformations, which can be used for analyzing and preserving liveness and boundedness.

- Fusion of Series Places (FSP).

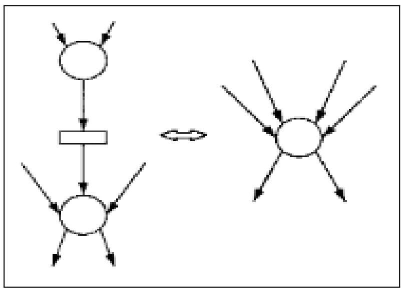


Figure 5.3: FSP.

- Fusion of Series Transitions (FST).

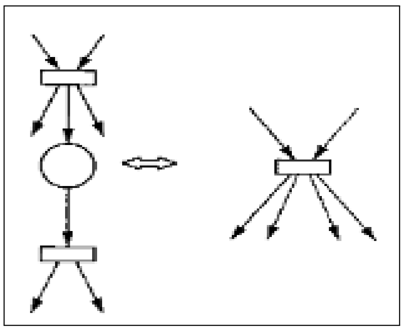


Figure 5.4: FST.

- Fusion of Parallel Places (FPP).

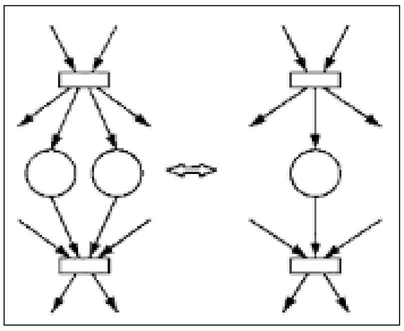


Figure 5.5: FPP.

- Fusion of Parallel Transitions (FPT).

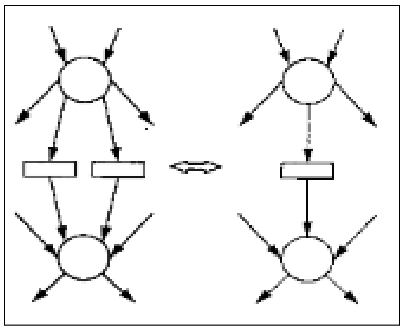


Figure 5.6: FPT.

But what about the specific properties relying on the linear structure of system?

Here we need to talk about temporal logic and exactly LTL as a mathematical language to express these kinds of properties.

5.3 Temporal logic

Temporal logic has been studied as a branch of logic for several decades. It was developed as a logical framework to formalize reasoning about time, and it could be a useful tool to formalize reasoning about the execution sequence of programs and especially of concurrent programs [35]. There is a several of algorithms used in temporal logic to specialize the set of properties that we want to verify, among them we cite : LTL and CTL.

- LTL (Linear Temporal Logic): Used when there is only one path in the future.
- CTL (Computational Tree Logic): We use it when the future is not determined, i.e there a there are different paths in the future, any one of which might be an actual path that is realized [36].

In our project we have only expressed the set of properties using LTL.

5.3.1 LTL operators

With LTL we can use the same propositional logic operators, in addition to some others to represent the time like :

- $F_p \equiv \diamond_p$: Eventually p or Somtimes p (In the future).
- $G_p \equiv \Box_p$: Always p.
- $X_p \equiv \bigcirc_p$: Next time p.
- $p \cup q$: p is true Until q being true.

5.4 Formal verification using model-checker

Model checker is a formal method for the verification of systemss behavior, developed by Clarke , Emerson, Queille and Sifakis in early 1980 [37].
Model checker offers the designer a still growing set of ready to use algorithms and techniques for the analysis of systems properties. So, it is potentially more acceptable from practitioners point of view. Generally, the approach consists in the following. First, out of some primary specification of system components (e.g. programs or some programs like notation, state diagrams or other finite state graphs

...etc) one has to build a large but finite graph containing all possible (reachable) system states and all possible transitions among them. This graph makes the (finite state) behavioral model of a system. Each path in the graph represents the allowable execution or the (part of) behavior of a system. The graph contains all possible executions or behaviors. Also, the property in question has to be specified. In a temporal model checking the property is specified as a temporal expression, i.e. the formula involving temporal operators (always, eventually, until, next) in addition to boolean ones (and, or, not) and the quantifiers (in all paths starting from state x, there exists a path starting from x such that...). Then, given the system model and the property, the exhaustive search of the systems state graph is performed, aimed to decide whether the desired property holds or not [38].
The verification process using model checker is resumed in the Figure 5.7.

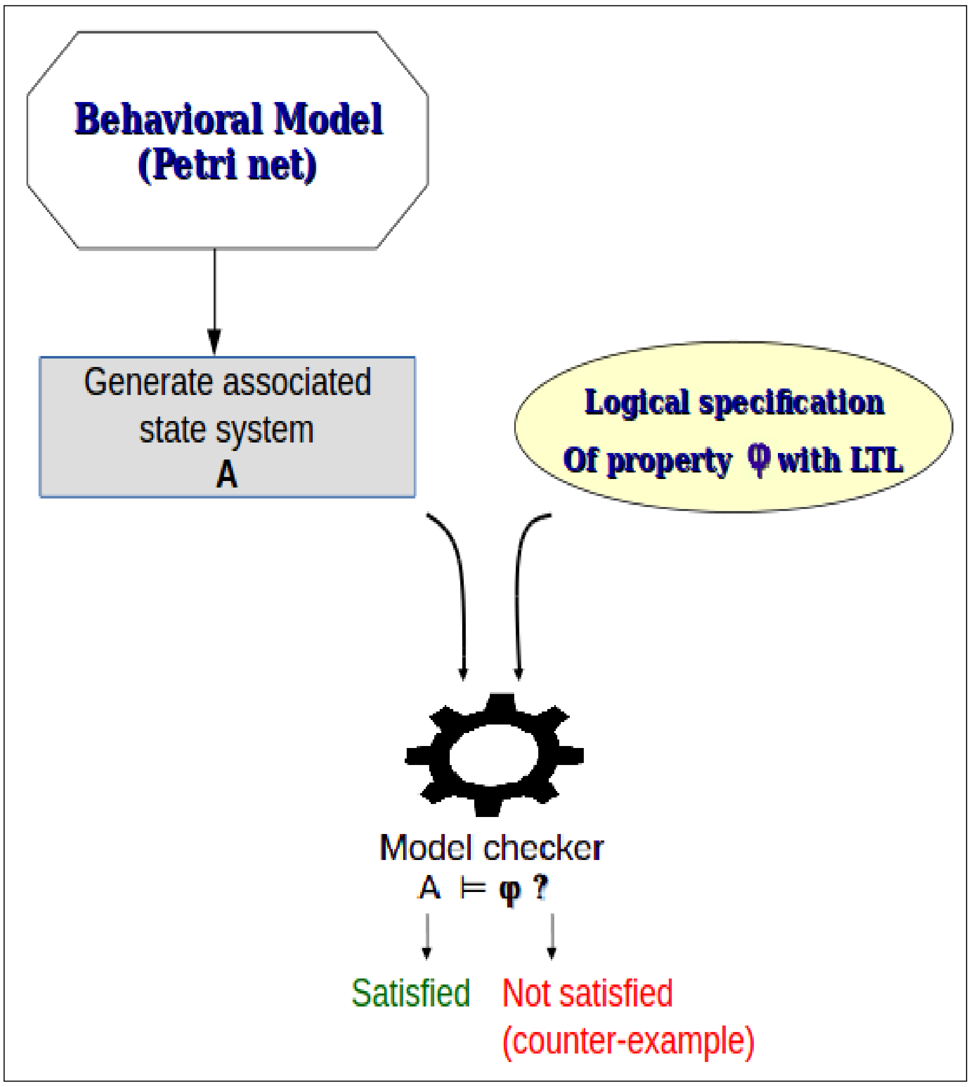


Figure 5.7: Model checker precess.

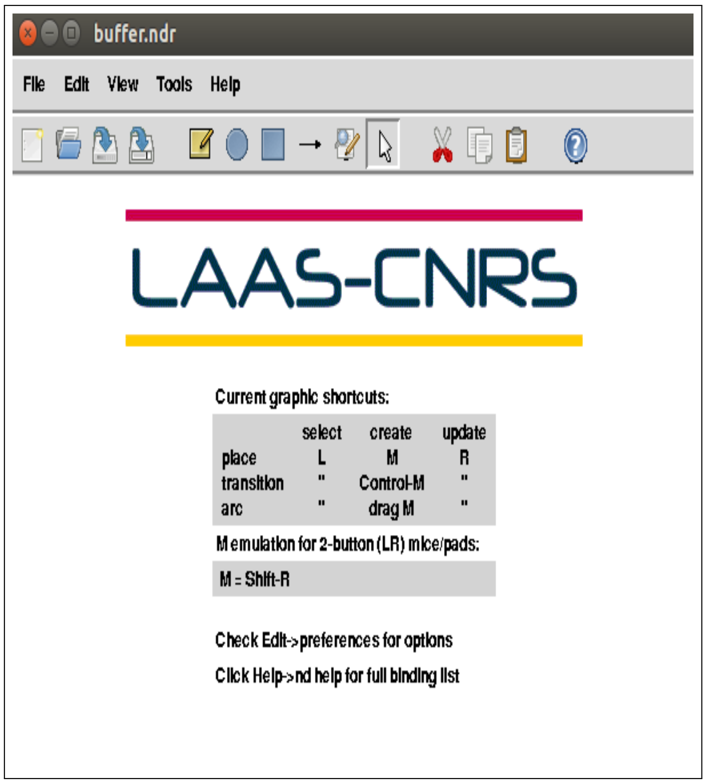
5.5 Formal verification tools

We distinguish between two kinds of tools, one of them used to model and verify the general properties of Petri net, and the other kind of tools is used to verify a logical properties using model checker.

5.5.1 Petri nets modeling tools

In this section we will try explain the most useful tools of verification in Petri net which are: TINA [39] and INA [40] tools.

TINA (Tima Petri Net Analyser)
TINA is a software environment to edit and analyze Petri nets and Time Petri nets, developed by the research groups of LAAS/CNRS.



Current graphic shortcuts:

	select	create	update
place	L	M	R
transition	"	Control-M	"
arc	"	drag M	"

M emulation for 2-button (LR) mice/pads:

M = Shift-R

Check Edit->preferences for options

Click Help->nd help for full binding list

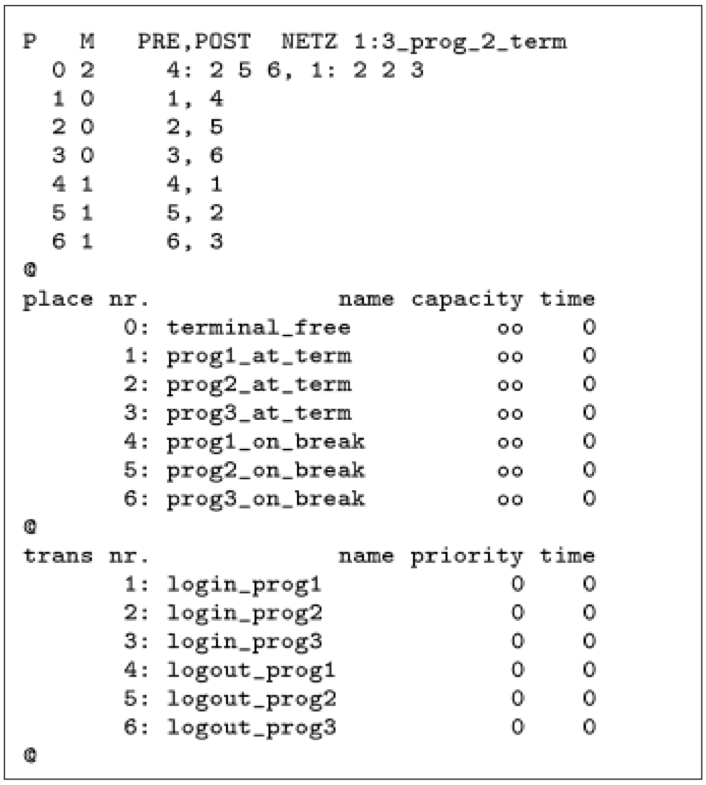
Figure 5.8: TINA screen snapchot

The TINA toolbox includes the following tools:

- nd (NetDraw): Editor and GUI for Petri nets, Time Petri Nets and Automata,
- tina: Construction of reachability graphs,

- sift: Construction and checking of reachability graphs,
- selt: A State/Event LTL model checker,
- ktzio: Conversion tool for Kripke transition systems,
- play: Stepper simulator.

INA (Integrated Net Analyser)
INA is an analytical tool widely used by the Petri nets community. It was developed by Prof. Dr. Peter H. Starke [41]. It is an interactive tool that allows the user to Reduce, execute and analyze Petri nets models with a textual form.
To analyze a Petri net with INA we should transform it to the acceptable textual description shows in Figure 5.9.



```
P   M   PRE,POST  NETZ 1:3_prog_2_term
  0 2     4: 2 5 6, 1: 2 2 3
  1 0     1, 4
  2 0     2, 5
  3 0     3, 6
  4 1     4, 1
  5 1     5, 2
  6 1     6, 3
@
place nr.             name capacity time
       0: terminal_free          oo    0
       1: prog1_at_term          oo    0
       2: prog2_at_term          oo    0
       3: prog3_at_term          oo    0
       4: prog1_on_break         oo    0
       5: prog2_on_break         oo    0
       6: prog3_on_break         oo    0
@
trans nr.             name priority time
       1: login_prog1           0     0
       2: login_prog2           0     0
       3: login_prog3           0     0
       4: logout_prog1          0     0
       5: logout_prog2          0     0
       6: logout_prog3          0     0
@
```

Figure 5.9: INA textual syntax.

5.5.2 Model checker tools

Here we present a comparison study between the several models checking tools extracted from [42], this comparison has been made following these criteria:

- Properties language : It means what are the mathematical language used to express the properties,

- GUI : Is the tool has a graphical user interface?,
- Counter example generation: In case when the verified properties is false, is the tool generate a counter example?.

Tool Name	Properties language	GUI	Counter example generation
SPIN [43]	LTL	Yes	Yes
SAL [44]	LTL	No	Yes
LTSA [45]	LTL	Yes	Yes
CADENCE SMV [46]	LTL, CTL	Yes	No
Expander2 [47]	CTL	NO	No

Table 5.1: Comparaison study between model checker tools.

To verify our Petri nets model we choose to use TINA tool for the following reasons:

- TINA has its own model checker which is *selt* (i.e without Tina we should use spin or any author model checker tools independently),
- It has a GUI which facilitate the Petri nets management,
- It has a GUI simulator in order to simulate the net growing.

Now after executing the transformation rules presented in chapter 4, the resulted Petri nets is not compatible with the syntax textual format of TINA tool, and to resolve that, it is necessary to define a transformation from ATL Petri net's format which is XML, to TINA textual format presented in Figure 5.11.

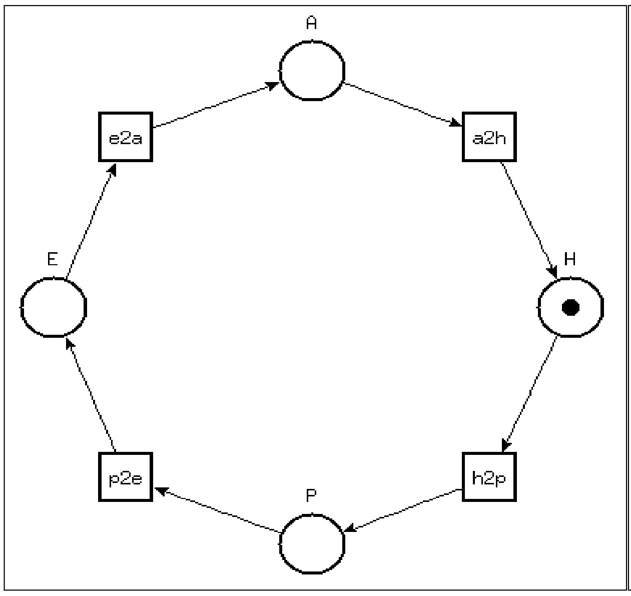
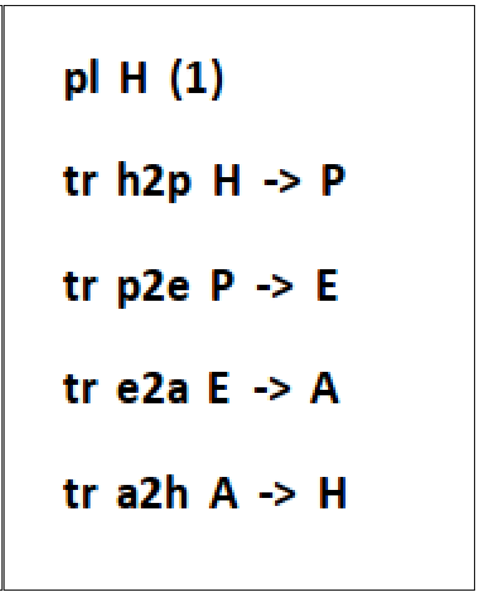


Figure 5.10: TINA .nd.



```
pl H (1)

tr h2p H -> P

tr p2e P -> E

tr e2a E -> A

tr a2h A -> H
```

Figure 5.11: TINA .net.

The transformation from ATL to TINA is illustrated by the following code.

```
query ATL2TINA =
   PN!PetriNets.allInstances()->asSequence() ->
       first().generatePetriNet().writeTo('/Rules/Rules/final.net');

helper context PN!PetriNets def : generatePetriNet() : String =
     let cpt : Integer = 0 in
     'net generatedPNFromTinaModelBE\n'+
     self.place->iterate(pl; accPL: String =''| accPL + pl.generatePlace()+'\n')+
     self.transition->iterate(tr ; accTR: String =''| accTR +
         tr.generateTransition()+'\n');

helper context PN!Place
def : generatePlace() : String = 'pl '+self.Name
                                 +if(self.NbrTokens<>0) then
                                     ('  ('+self.NbrTokens.toString()+')\n')
                                     else '\n' endif;
helper context PN!Transition
def : generateTransition() : String = 'tr '+self.Name+' '+
   PN!ArcP2T.allInstances()->select(a1 | a1.target = self)->iterate(a1; ac1: String
       = ''| ac1+a1.source.Name+'*'+a1.Nbr.toString()+' ')
   +' -> '+
   PN!ArcT2P.allInstances()->select(a1 | a1.source = self)->iterate(a1; ac1: String
       = ''| ac1+a1.target.Name+'*'+a1.Nbr.toString()+' ')
   +' \n ';
```

5.6 Conclusion

The main points presented in this chapter are :

- Petri nets general properties such as : reachability, livennes, boundedness, reversibility,
- The formal verification techniques of Petri nets, which are : coverability tree, algebraic and reduction methods,
- The process of verification using model checking and the specification of properties using LTL,
- An ATL query to transform a Petri net represented with XML format to textual format of TINA tool.

And like that the process of formal verification which is the last step of our proposed approach is terminated, In the next chapter we will try to present some results by exposing three cases study.

Chapter 6

Case Study and Experimental Results

6.1 Introduction

In this chapter we will try to execute our approach in three different systems, in which they have a different degree of complexity (simple, medium, complex) which are : traffic light, elevator and online test systems, and for each one we will use one of the verification techniques of Petri nets presented in Chapter 5 such as : coverability tree, algebraic methods and reduction rules using TINA tool.

6.2 Example 1 : Traffic light system

A traffic light system is composed from three colors : green, red and orange. That means that every object within this system can being in three states: have the green color, the red color or the orange color.

Figure 6.1: Traffic light system.

The first step is to modulate the statechart diagram using our GMF editor.

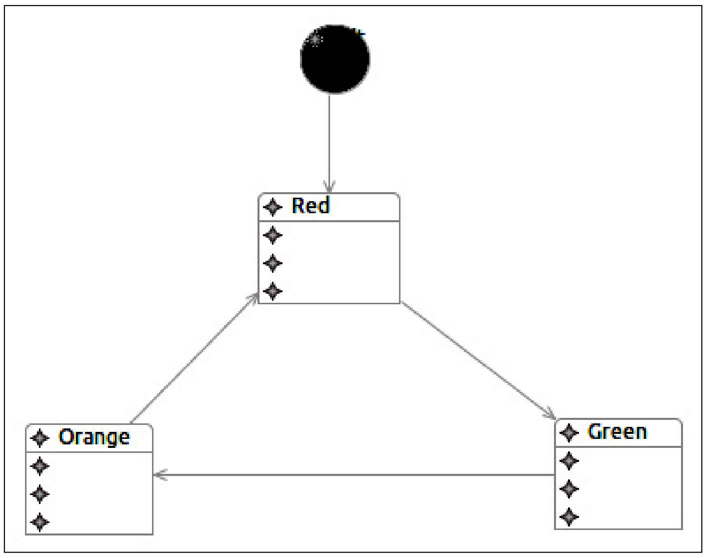


Figure 6.2: Trafic light system state diagram.

After executing the two defined grammars, the equivalent Petri nets will be generated.

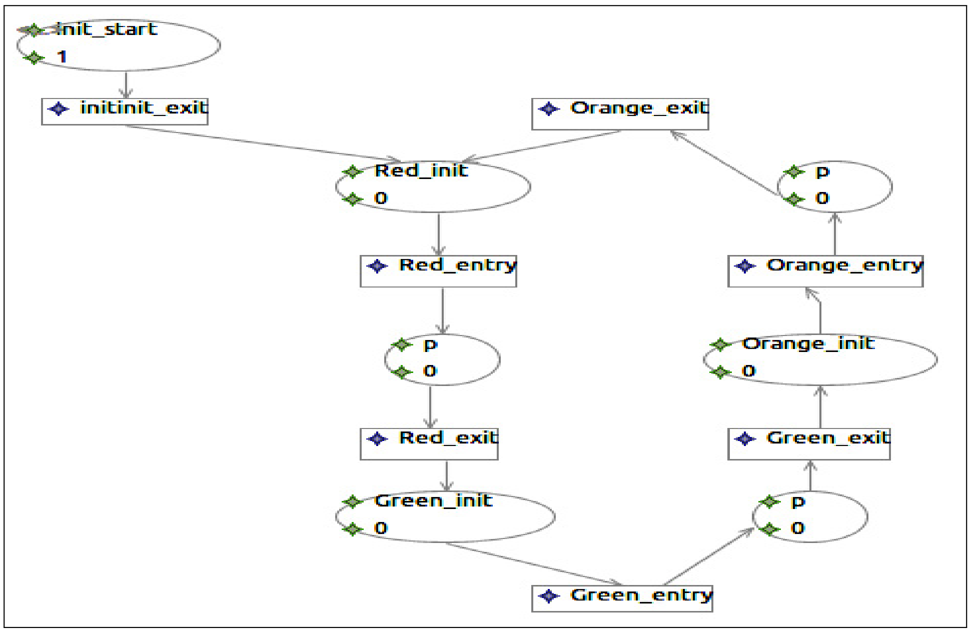
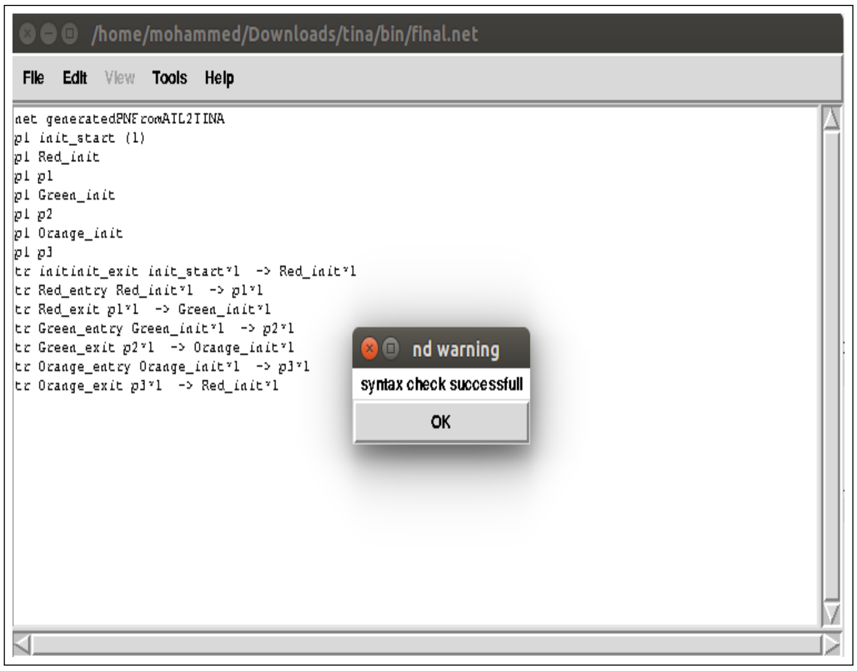


Figure 6.3: Trafic light system generated petri net.

To get the possibility to import the resulted Petri nets on TINA tools, we need to execute the transformation's grammar ATL2TINA. The result is a generated file with *.net* extension, its content is the following code.

```
net generatedPNFromATL2TINA
pl init_start (1)
pl Red_init
pl p1
pl Green_init
pl p2
pl Orange_init
pl p3
tr initinit_exit init_start*1 -> Red_init*1
tr Red_entry Red_init*1 -> p1*1
tr Red_exit p1*1 -> Green_init*1
tr Green_entry Green_init*1 -> p2*1
tr Green_exit p2*1 -> Orange_init*1
tr Orange_entry Orange_init*1 -> p3*1
tr Orange_exit p3*1 -> Red_init*1
```

Now we can import it with TINA tool.
The next step is to verify the correctness of the syntax by just clicking on *Tools* then *check net*.



```
net generatedPNFromATL2TINA
pl init_start (1)
pl Red_init
pl p1
pl Green_init
pl p2
pl Orange_init
pl p3
tr initinit_exit init_start*1  -> Red_init*1
tr Red_entry Red_init*1  -> p1*1
tr Red_exit p1*1  -> Green_init*1
tr Green_entry Green_init*1  -> p2*1
tr Green_exit p2*1  -> Orange_init*1
tr Orange_entry Orange_init*1  -> p3*1
tr Orange_exit p3*1  -> Red_init*1
```

syntax check successfull

Figure 6.4: Tina syntax verification *-example 1-*.

As seen in the Figure 6.4 the syntax is successfully checked.
Now as we said, TINA has a graphical editor for Petri nets, and to use it we need to click on *edit* then *draw* and choosing one of the available editors. In our case we have used *circo drawing tools*, the result is illustrated in the Figure 6.5.

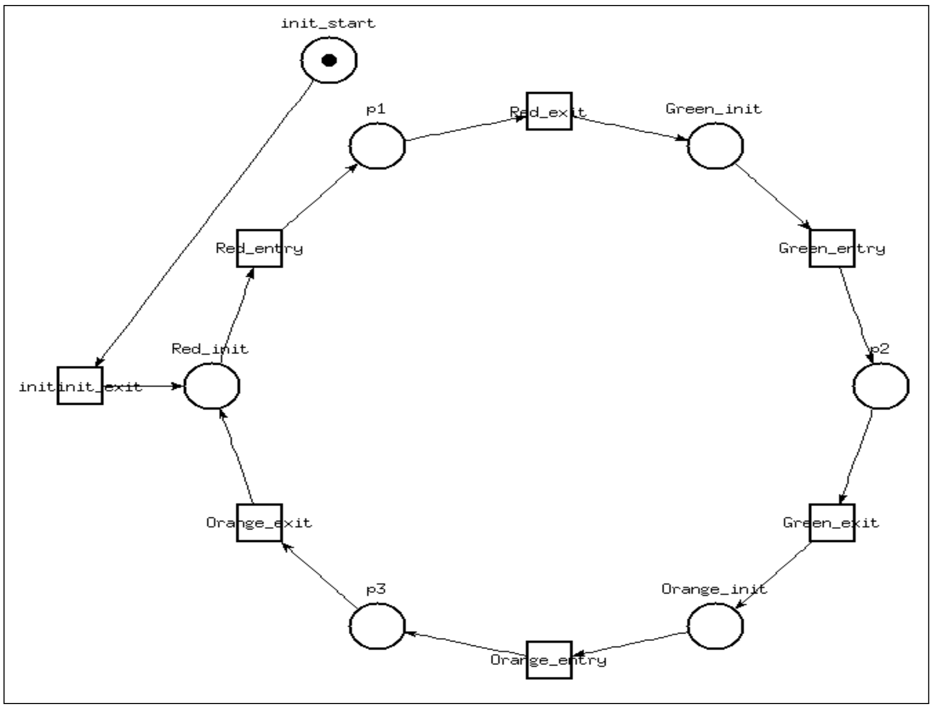


Figure 6.5: Circo drawing tool.

TINA also give the possibility to follow the growing of the net using its own stepper simulator.

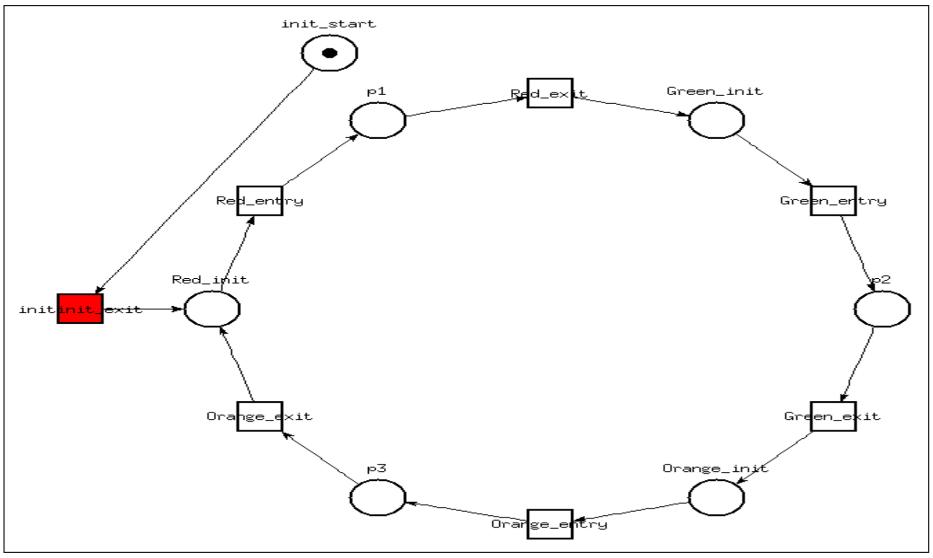
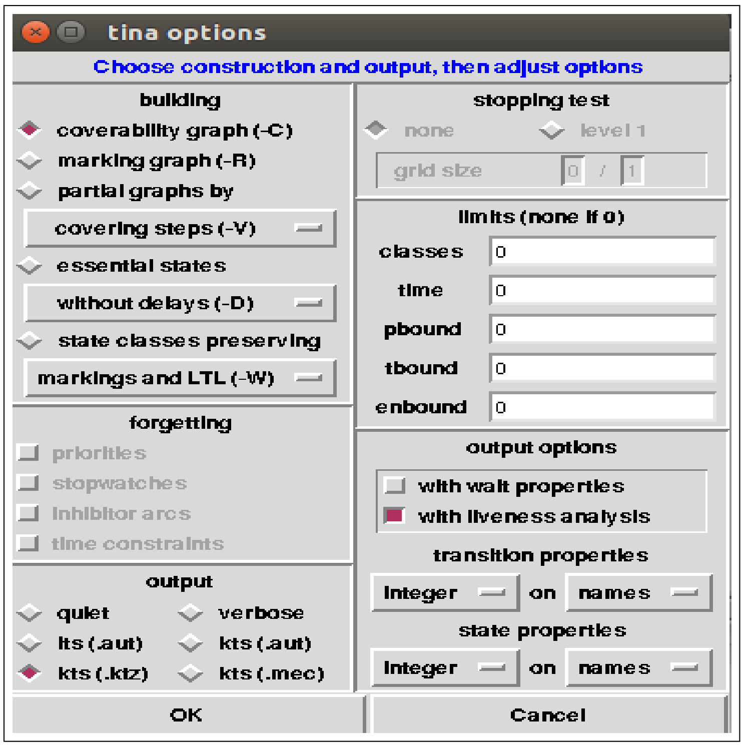


Figure 6.6: Stepper simulator.

Now to verify the general properties of our Petri net model we need to click on *tools* then *reachability analysis* and the window presented in Figure 6.7 will be displayed.



tina options

Choose construction and output, then adjust options

building
- [✓ coverability graph (-C)]
- marking graph (-R)
- partial graphs by
  - covering steps (-V)
- essential states
  - without delays (-D)
- state classes preserving
  - markings and LTL (-W)

forgetting
- priorities
- stopwatches
- inhibitor arcs
- time constraints

output
- quiet
- verbose
- lts (.aut)
- kts (.aut)
- [✓ kts (.ktz)]
- kts (.mec)

stopping test
- none
- level 1
- grid size 0 / 1

limits (none if 0)

classes	0
time	0
pbound	0
tbound	0
enbound	0

output options
- with wait properties
- [✓ with liveness analysis]

transition properties: integer on names

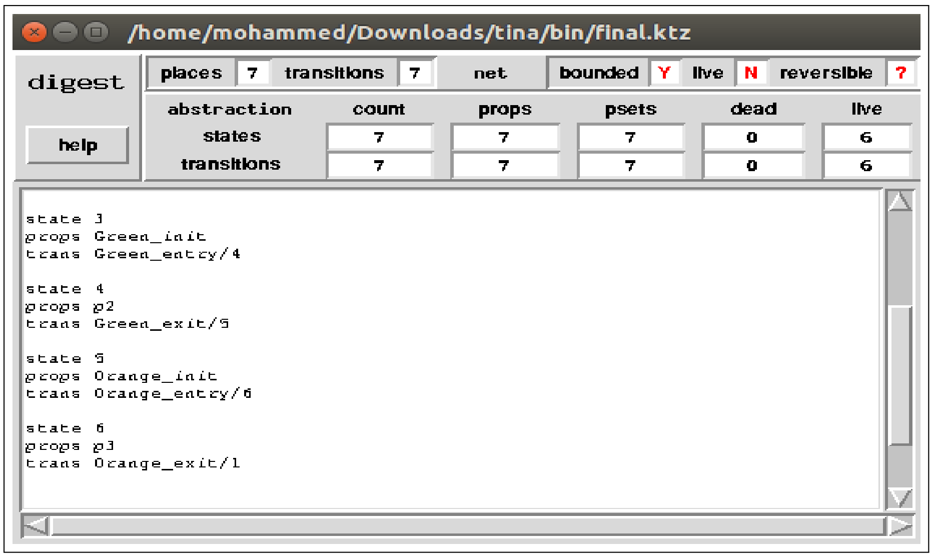
state properties: integer on names

OK Cancel

Figure 6.7: TINA verification properties *-example 1-*.

- Use the coverability graph technique,
- In *"output properties"* section, select *"with liveness analysis"* in order to check the liveness properties,
- In *"output section"*, select *"ktz(.ktz)"* with the goal to generate automatically the kripke transition system correspond to this net, in order to get the possibility to verify an LTL properties late.

The result is illustrated in the Figure 6.8.


/home/mohammed/Downloads/tina/bin/final.ktz

digest

help

places	7	transitions	7	net	bounded	Y	live	N	reversible	?

abstraction	count	props	psets	dead	live
states	7	7	7	0	6
transitions	7	7	7	0	6

```
state 3
props Green_init
trans Green_entry/4

state 4
props p2
trans Green_exit/5

state 5
props Orange_init
trans Orange_entry/6

state 6
props p3
trans Orange_exit/1
```

Figure 6.8: Result of analysis -*example 1*-.

- The net is composed from 7 places and 7 transitions,
- The net is bounded because each place in the net can have at most one token,
- The net is not live because as seen, not all the transitions are live,
- the net is not reversible.

We can also verify some author properties expressed by LTL like the following:

- Always there will be a red color,

```
[] <> (Red_init);
```

- Always there will be a green color,

```
[] <> (Green_init);
```

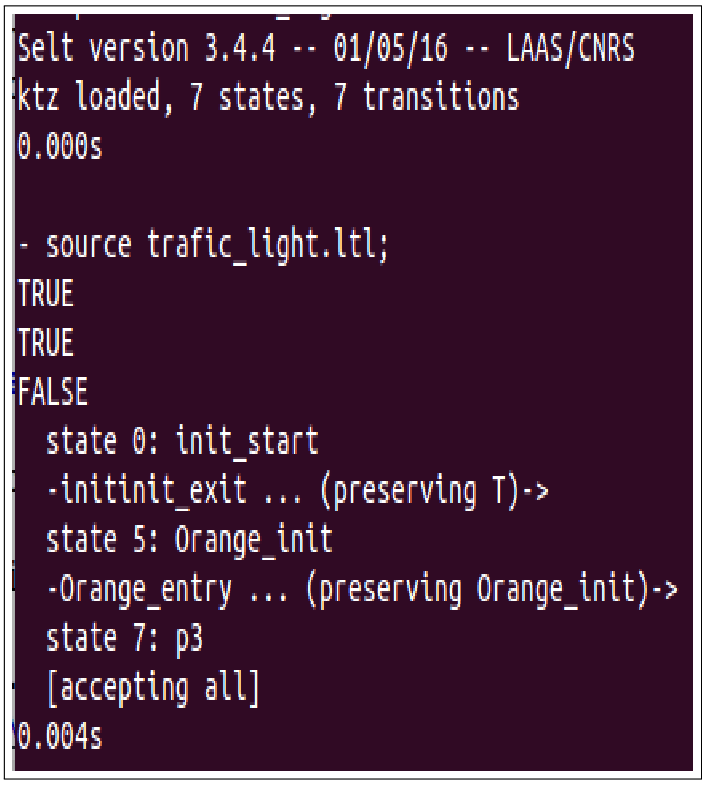
- The Orange color will never being ON.

```
- <> (Orange_init);
```

To check the previous properties we need to put them on a file and save it with *.ltl* extension, then executing the following command line :

```
./selt -S final.scs final.ktz -prelude trafic_light.ltl
```

It means execute model checker *selt* in which it has as input the kripke transition system with *.ktz* extension and the set of LTL properties, and give as result true if the property is checked, and false with a counter-example if the property is not verified. The counter-example will be setted in the file *final.scs*
The result is presented in the Figure 6.9.



```
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 7 states, 7 transitions
0.000s

- source trafic_light.ltl;
TRUE
TRUE
FALSE
  state 0: init_start
  -initinit_exit ... (preserving T)->
  state 5: Orange_init
  -Orange_entry ... (preserving Orange_init)->
  state 7: p3
  [accepting all]
0.004s
```

Figure 6.9: Result of LTL analysis -*example 1*-.

As seen in Figure 6.9, the two first LTL properties are true, and the last is false. In this case TINA has generated a counter-example into *final.scn* file like the following.

```
initinit_exit Red_entry Red_exit Green_entry Green_exit Orange_entry
# accepting all
```

6.3 Example 2 :Elevator system

An elevator system can being in 4 states: opened, closed, started and stopped.

Figure 6.10: Elevator system.

The statechart diagram of this system is displayed in the Figure 6.11

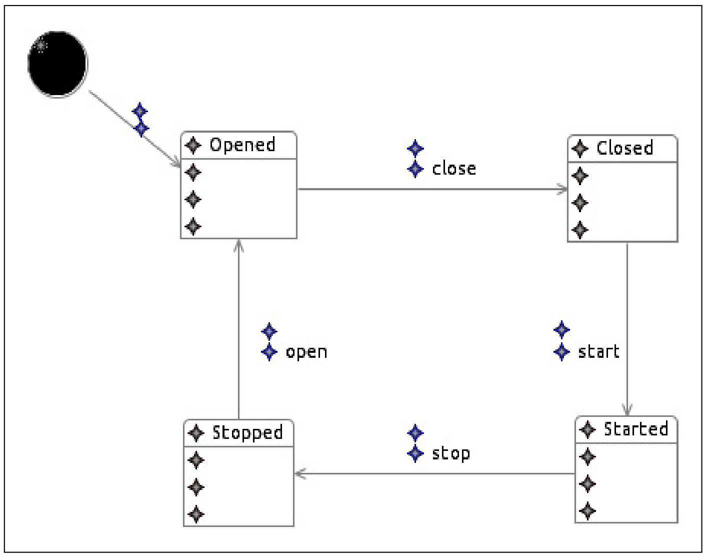


Figure 6.11: Elevator system state diagram.

After executing the two defined grammars, the equivalent Petri nets will be generated.

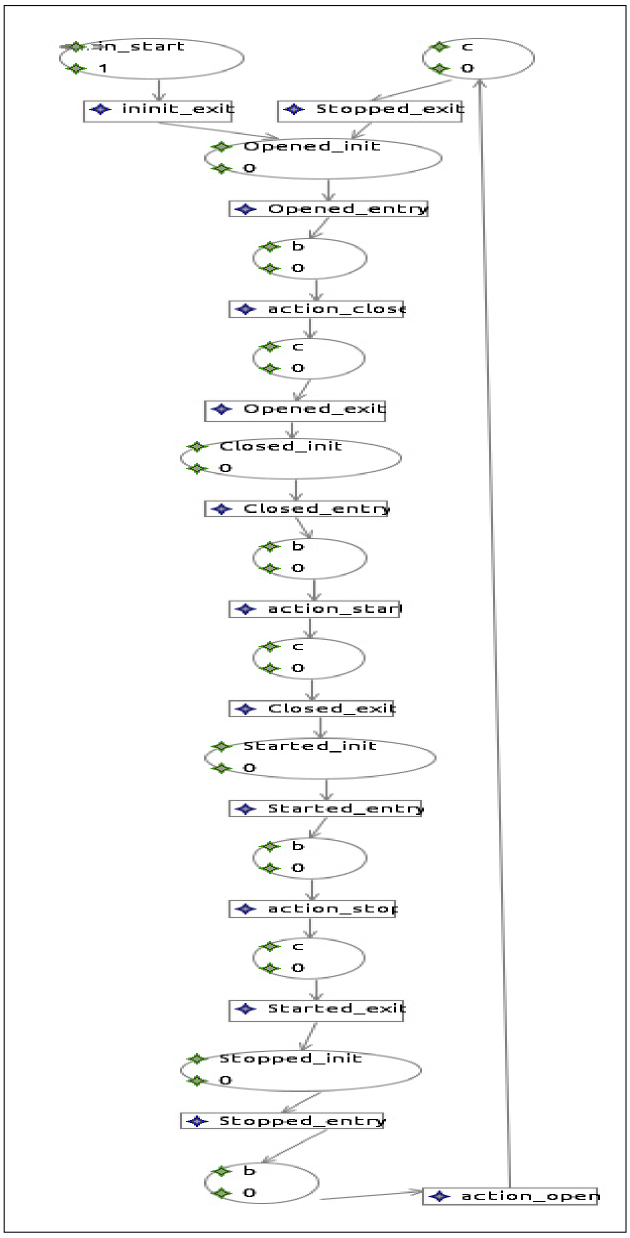


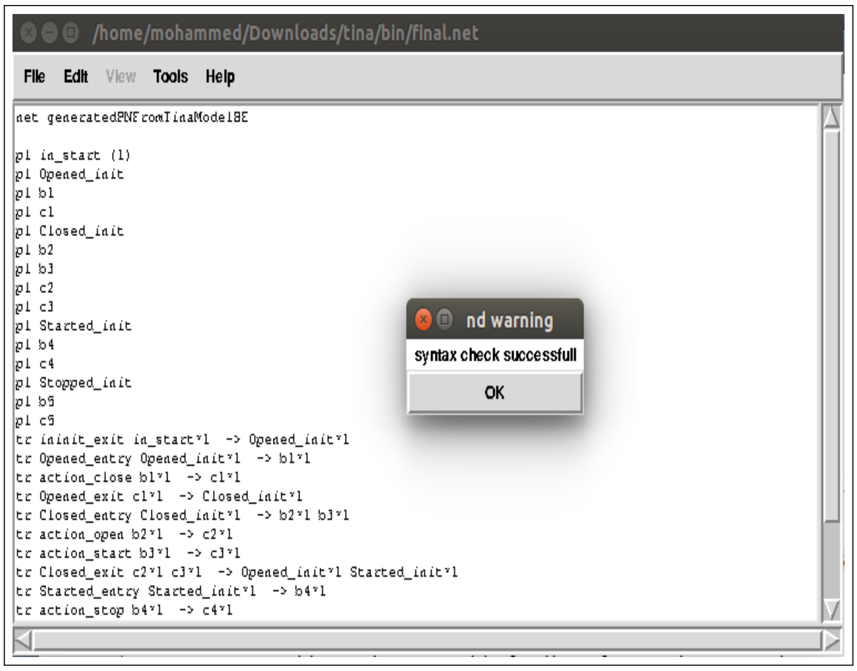
Figure 6.12: Elevator system generated petri net.

To get the possibility to import the resulted Petri nets on TINA tools, we need to execute the transformation's grammar ATL2TINA. The result is a generated file with *.net* extension, its content is the following code.

```
net generatedPNFromTinaModelBE

pl in_start (1)
pl Opened_init
pl b1
pl c1
pl Closed_init
pl b2
pl c2
pl Started_init
pl b3
pl c3
pl Stopped_init
pl b4
pl c4
tr ininit_exit in_start*1 -> Opened_init*1
tr Opened_entry Opened_init*1 -> b1*1
tr action_close b1*1 -> c1*1
tr Opened_exit c1*1 -> Closed_init*1
tr Closed_entry Closed_init*1 -> b2*1
tr action_start b2*1 -> c2*1
tr Closed_exit c2*1 -> Started_init*1
tr Started_entry Started_init*1 -> b3*1
tr action_stop b3*1 -> c3*1
tr Started_exit c3*1 -> Stopped_init*1
tr Stopped_entry Stopped_init*1 -> b4*1
tr action_open b4*1 -> c4*1
tr Stopped_exit c4*1 -> Opened_init*1
```

Now we can import it with TINA tool.
The next step is to verify the correctness of the syntax by just clicking on *Tools* then *check net*.


/home/mohammed/Downloads/tina/bin/final.net
File Edit View Tools Help
nd warning
OK

```
net generatedPNFromTinaModelBE

pl in_start (1)
pl Opened_init
pl b1
pl c1
pl Closed_init
pl b2
pl b3
pl c2
pl c3
pl Started_init
pl b4
pl c4
pl Stopped_init
pl b5
pl c5
tr ininit_exit in_start*1  -> Opened_init*1
tr Opened_entry Opened_init*1  -> b1*1
tr action_close b1*1  -> c1*1
tr Opened_exit c1*1  -> Closed_init*1
tr Closed_entry Closed_init*1  -> b2*1 b3*1
tr action_open b2*1  -> c2*1
tr action_start b3*1  -> c3*1
tr Closed_exit c2*1 c3*1  -> Opened_init*1 Started_init*1
tr Started_entry Started_init*1  -> b4*1
tr action_stop b4*1  -> c4*1
```

syntax check successfull

Figure 6.13: Tina syntax verification -*example 2*-.

As seen in the Figure 6.13 the syntax is successfully checked.

Now as we said TINA has a graphical editor for Petri nets, and to use it we need to click on *edit* then *draw* and choosing one of the available editors.
In the first example we have used *circo drawing tools*, but in this example we will try to use *neato drawing tools* as illustrated in Figure 6.14.

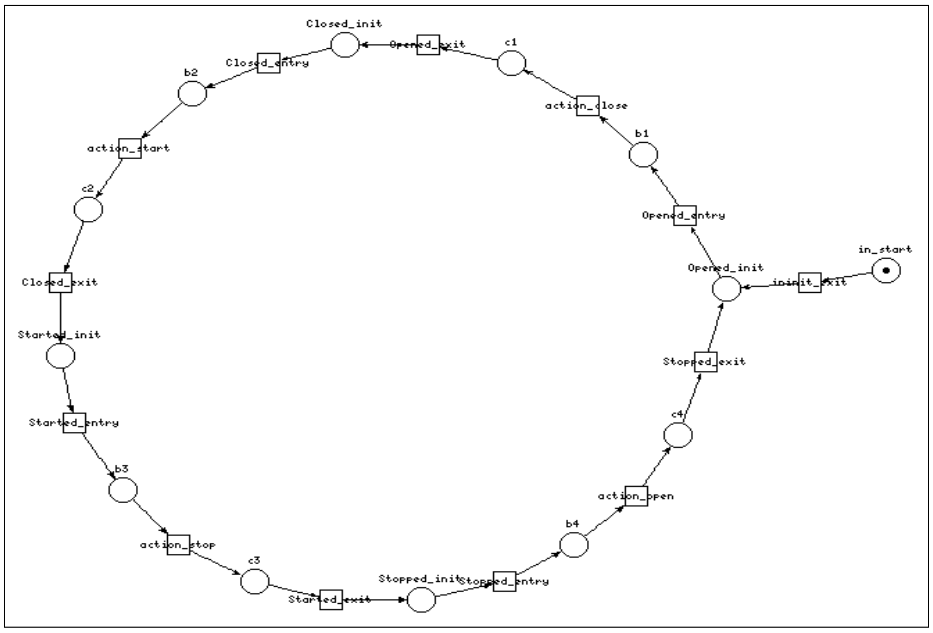
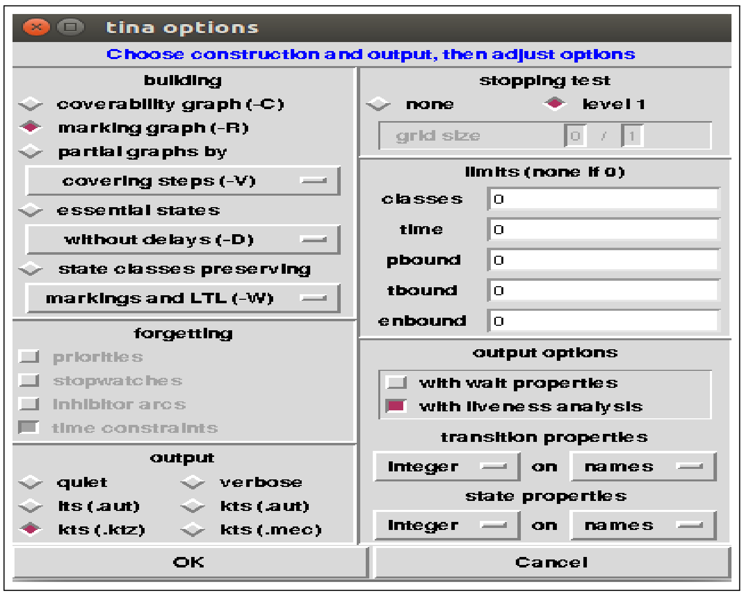


Figure 6.14: Neato drawing tool.

Now to verify the structural properties of our Petri net model we need to click on *tools* then *marking analysis* and the window presented in Figure 6.15 will be displayed.


tina options

Choose construction and output, then adjust options

building
- coverability graph (-C)
- [✓ marking graph (-R)]
- partial graphs by
  - covering steps (-V)
- essential states
  - without delays (-D)
- state classes preserving
  - markings and LTL (-W)

forgetting
- priorities
- stopwatches
- inhibitor arcs
- time constraints

output
- quiet — verbose
- lts (.aut) — kts (.aut)
- [✓ kts (.ktz)] — kts (.mec)

stopping test
- none — [✓ level 1]
- grid size 0 / 1

limits (none if 0)

classes	0
time	0
pbound	0
tbound	0
enbound	0

output options
- with wait properties
- [✓ with liveness analysis]

transition properties
integer on names

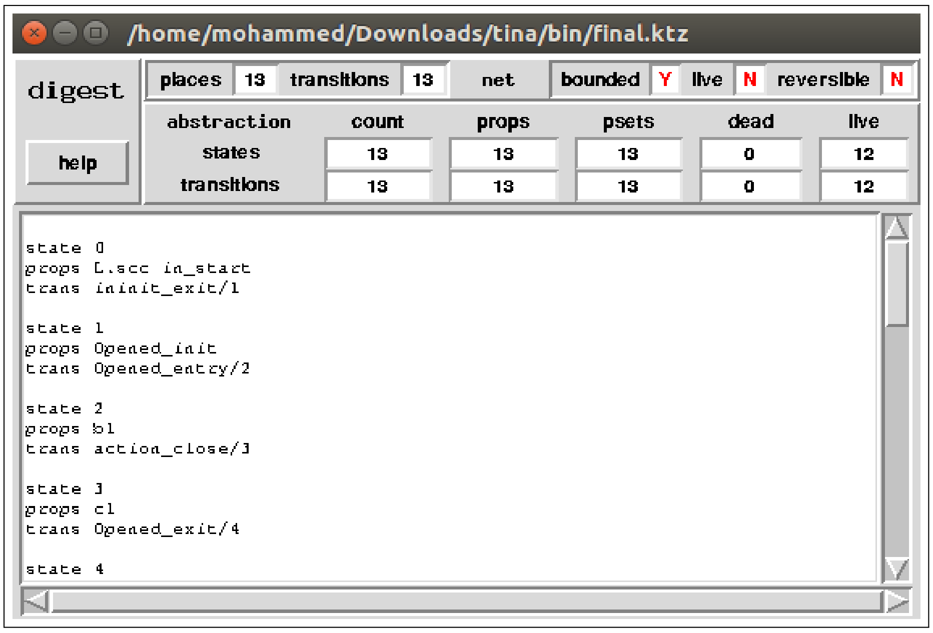
state properties
integer on names

OK — Cancel

Figure 6.15: TINA verification properties -*example 2*-.

- In the first example we have used the coverability graph technique, now we will try to use algebraic methods by selecting ”marking graph”.
- In *”output properties”* section, select *”with liveness analysis”* in order to check the liveness properties.
- In *”output section”*, select *”ktz(.ktz)”* with the goal to generate automatically the kripke transition system correspond to this net, in order to get the possibility to verifying an LTL properties lately.

The result is displayed in Figure 6.16.


/home/mohammed/Downloads/tina/bin/final.ktz

digest

help

places	13	transitions	13	net	bounded	Y	live	N	reversible	N

abstraction	count	props	psets	dead	live
states	13	13	13	0	12
transitions	13	13	13	0	12

```
state 0
props L.scc in_start
trans inint_exit/1

state 1
props Opened_init
trans Opened_entry/2

state 2
props b1
trans action_close/3

state 3
props c1
trans Opened_exit/4

state 4
```

Figure 6.16: Result of analysis -*example 2*-.

- The net is composed from 13 places and 13 transitions,
- The net is bounded because all the places in the net can have at most one token,
- The net is not live because not all the transitions are live,
- The net is not reversible.

We can also verify some author properties expressed by LTL like the following:

- Always the elevator will never be opened until being stopped,

```
        [] <> (-(Opened_init) U action_stop);
```

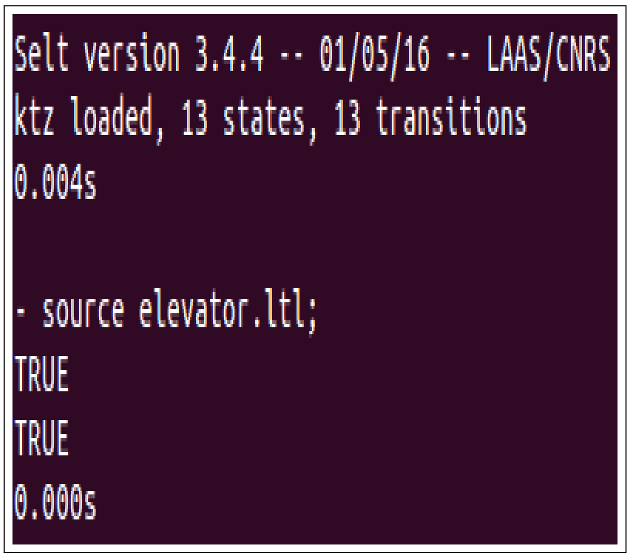
- Always the elevator will never be started until the door being closed.

```
        [] <> (-(Started_init) U action_close);
```

After putting them on an *.ltl* extension file, and executing the following command line :

```
    ./selt -S final.scs final.ktz -prelude trafic_light.ltl
```

The result will be presented like the Figure 6.17.



```
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 13 states, 13 transitions
0.004s

- source elevator.ltl;
TRUE
TRUE
0.000s
```

Figure 6.17: Result of LTL analysis -*example 2*-.

As seen all the LTL properties are true.

6.4 Example 3 : Online test system

In this example we will represent the user web's states once connecting to any server for passing a test online.
Once the user connect, he will receive the questions. While answering he passes from level 1 until level 3. Once the test is terminated the user passes to refused or accepted state.
The first step is to modulate the statechart diagram using our GMF editor.

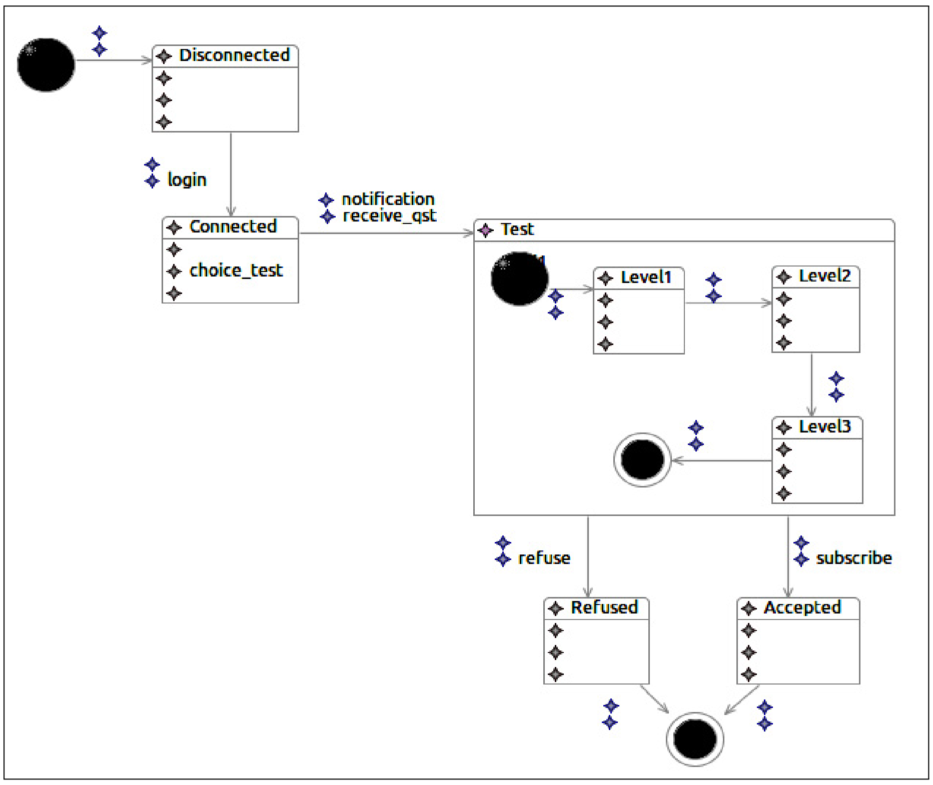


Figure 6.18: User state diagram.

The equivalent Petri net will be generated after executing the two defined grammars.

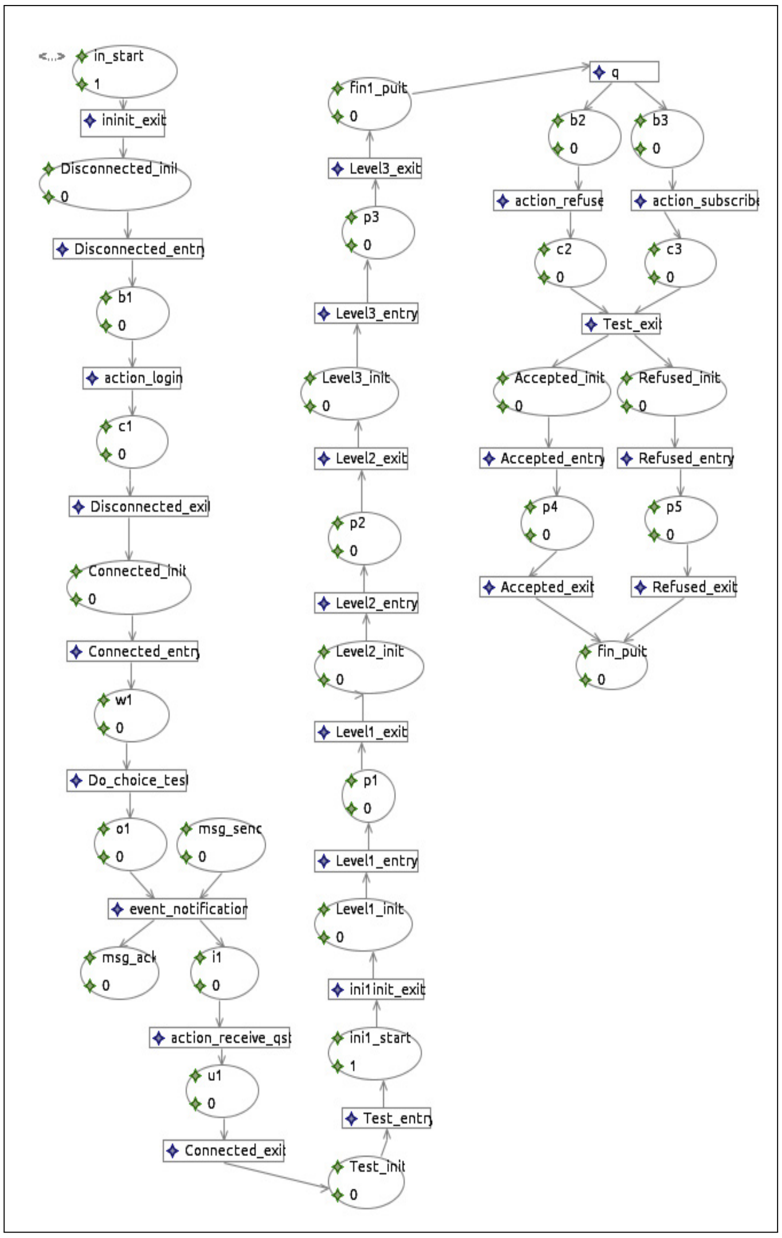


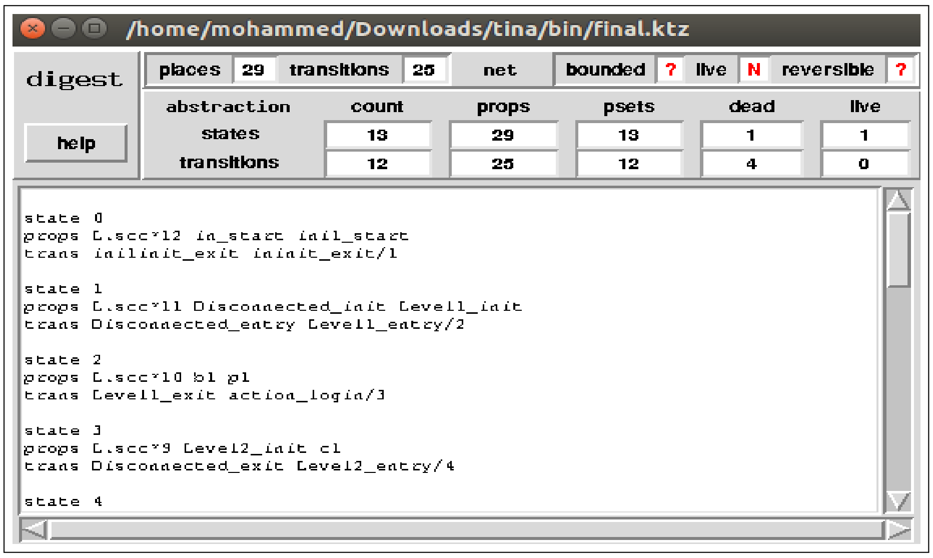
Figure 6.19: Online test system generated petri net.

Like the previous examples, we execute now the ATL2TINA grammar.

```
net generatedPNFromTinaModelBE

pl in_start (1)
pl ini1_start (1)
pl Disconnected_init
pl b
pl c
pl Level1_init
pl p1
pl Level2_init
pl p2
pl Level3_init
pl p3
pl Accepted_init
pl p4
pl Refused_init
pl p5
pl Connected_init
pl w
pl o
pl msg_send
pl msg_ack
pl i
pl u
pl fin1_puit
pl fin_puit
pl Test_init
pl b1
pl b2
pl c1
pl c2
tr ininit_exit in_start*1 -> Disconnected_init*1
tr ini1init_exit ini1_start*1 -> Level1_init*1
tr Disconnected_entry Disconnected_init*1 -> b*1
tr action_login b*1 -> c*1
tr Disconnected_exit c*1 -> Connected_init*1
tr Level1_entry Level1_init*1 -> p1*1
tr Level1_exit p1*1 -> Level2_init*1
tr Level2_entry Level2_init*1 -> p2*1
tr Level2_exit p2*1 -> Level3_init*1
tr Level3_entry Level3_init*1 -> p3*1
tr Level3_exit p3*1 -> fin1_puit*1
tr Accepted_entry Accepted_init*1 -> p4*1
tr Accepted_exit p4*1 -> fin_puit*1
tr Refused_entry Refused_init*1 -> p5*1
tr Refused_exit p5*1 -> fin_puit*1
tr Connected_entry Connected_init*1 -> w*1
tr Do_choice_test w*1 -> o*1
tr event_notification o*1 msg_send*1 -> msg_ack*1 i*1
tr action_receive_qst i*1 -> u*1
tr Connected_exit u*1 -> Test_init*1
tr Test_entry Test_init*2 -> ini1_start*1
tr q fin1_puit*1 -> b1*1 b2*1
tr action_refuse b1*1 -> c1*1
tr action_subscribe b2*1 -> c2*1
tr Test_exit c1*1 c2*1 -> Refused_init*1 Accepted_init*1
```

With the same way using *"partial graph"* technique, TINA give us the following results:


/home/mohammed/Downloads/tina/bin/final.ktz

places	29	transitions	25	net	bounded	?	live	N	reversible	?

abstraction	count	props	psets	dead	live
states	13	29	13	1	1
transitions	12	25	12	4	0

```
state 0
props L.scc*12 in_start init_start
trans initinit_exit ininit_exit/1

state 1
props L.scc*11 Disconnected_init Level1_init
trans Disconnected_entry Level1_entry/2

state 2
props L.scc*10 b1 p1
trans Level1_exit action_login/3

state 3
props L.scc*9 Level2_init c1
trans Disconnected_exit Level2_entry/4

state 4
```

Figure 6.20: Result of analysis *-example 3-*.

- The net is composed from 29 places and 25 transitions,
- The net is not live because there a deadlock situation,
- There is one dead place and 4 unfirable transitions.

Note: partial graph technique allow us only to check the liveness properties.

6.5 Conclusion

What has been presented in this chapter, is three case study with the goal to see the practical side of how to execute the defined transformation's rules and to get a Petri nets model from an input statechart diagram, and how to use TINA tool to verify a several types of properties in Petri net, such as : livenness, boundness, reversibility and LTL properties.

Conclusion and Future works

Through our project, we demonstrate an approach to verify UML statecharts diagrams by transforming them into Petri nets, based on the main concepts of model transformation defined in the domain of MDE such as: meta-models and the standard Atlas Transformation Language tool.
This approach has been developed according to the following steps :

- Define statecharts meta-model in order to establish any statechart source model composed from : initial, final, simple, composed states, and join and fork nodes,
- Define Petri nets meta-model with the goal to get a target a Petri net model that conforms to it,
- To create the statecharts and Petri nets diagrams editors, use Graphical Modeling Framework to define the graphical representation of the composed elements,
- Implement the set of rules to transform each node in the source model, to its equivalent on the target model,
- Use TINA tool to verify the general properties according to any Petri nets such as: livenness, reachability ...etc,
- Express the temporal properties using LTL, and 'selt' model checker to verify them.

As future work, we aim to complete the transformation of all the statecharts nodes that has been added in UML 2.5 version such as: choice node, history state ...etc, and develop a framework to facilitate the managing of the different steps with the goal to get a fully automatic approach.

References

[1] G. Booch, I. Jacobson and J. Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[2] D'Ambrogio, A. (2005, July). A model transformation framework for the automated building of performance models from UML models. In Proceedings of the 5th international workshop on Software and performance(pp. 75-86). ACM.

[3] Lopez-Grao, J. P., Merseguer, J., Campos, J. (2004). From UML activity diagrams to Stochastic Petri nets: application to software performance engineering. ACM SIGSOFT software engineering notes, 29(1), 25-36.

[4] Baresi, L.,Pezze, M. (2001). On formalizing UML with high-level Petri nets. In Concurrent object-oriented programming and petri nets (pp. 276-304). Springer Berlin Heidelberg.

[5] Ribeiro, . R. S. F.,Fernandes, J. M. (2006). Some rules to transform sequence diagrams into coloured Petri nets. In 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006) (pp. 237-256).

[6] Farooq, U., Lam, C. P., Li, H. (2007). Transformation methodology for UML 2.0 activity diagram into colored Petri nets. Advances in Computer Science and Technology, 128-133.

[7] Storrle, H. (2005). Semantics and verification of data flow in UML 2.0 activities. Electronic Notes in Theoretical Computer Science, 127(4), 35-52.

[8] Bouarioua, M., Chemaa, S., Chaoui, A. Transformation des diagrammes dtats-transitions UML vers les rseaux de Petri stochastiques gnraliss en utilisant la transformation de graphes.

[9] Harel, D. (1987). Statecharts: A visual formalism for complex systems. Science of computer programming, 8(3), 231-274.

[10] http://www.omg.org/spec/UML/2.5/PDF/

[11] Petri, C. A., Reisig, W. (2008). Petri net. Scholarpedia, 3(4), 6477.

[12] Murata, T. (1989). Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4), 541-580.

[13] Petri, C. A. (1962). Kommunikation mit automaten.

[14] Girault, C., Valk, R. (2013). Petri nets for systems engineering: a guide to modeling, verification, and applications. Springer Science and Business Media.

[15] David, R., Alla, H. (2010). Discrete, continuous, and hybrid Petri nets. Springer Science and Business Media.

[16] Bause, F., Kritzinger, P. S. (2002). Stochastic Petri Nets (Vol. 1, p. 0). Wiesbaden: Vieweg.

[17] Suzuki, I., Lu, H. (1989). Temporal Petri nets and their application to modeling and analysis of a handshake daisy chain arbiter. IEEE Transactions on Computers, 38(5), 696-704.

[18] Jensen, K. (1987). Coloured petri nets. In Petri nets: central models and their properties (pp. 248-299). Springer Berlin Heidelberg.

[19] Bézivin, J. (2004). In search of a basic principle for model driven engineering. Novatica Journal, Special Issue, 5(2), 21-24.

[20] Czarnecki, K., Helsen, S. (2003, October). Classification of model transformation approaches. In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture (Vol. 45, No. 3, pp. 1-17).

[21] Chared, Z., Tyszberowicz, S. S. (2013). Projective Template-Based Code Generation. In CAiSE Forum (pp. 81-87).

[22] https://eclipse.org/viatra/

[23] http://atom3.cs.mcgill.ca/

[24] http://www.user.tu-berlin.de/o.runge/agg/

[25] http://www.eclipse.org/atl/

[26] http://www.isis.vanderbilt.edu/tools/GReAT

[27] De Lara, J., Vangheluwe, H. (2002, April). AToM3: A Tool for Multi-formalism and Meta-modelling. In International Conference on Fundamental Approaches to Software Engineering (pp. 174-188). Springer Berlin Heidelberg.

[28] Taentzer, G. (1999, September). AGG: A tool environment for algebraic graph transformation. In International Workshop on Applications of Graph Transformations with Industrial Relevance (pp. 481-488). Springer Berlin Heidelberg.

[29] Projet Atlas, I. N. R. I. A. (2005). ATL: Atlas Transformation Language.

[30] http://www.omg.org/mda/mda_files/b+m_OMGCommittment.pdf

[31] http://andromda.sourceforge.net/

[32] http://www.omg.org/mda/mda_files/ArcStyler5_Whitepaper_220205.pdf

[33] https://eclipse.org/modeling/emf/

[34] https://www.eclipse.org/modeling/gmp/

[35] Wolper, P. (1985). The tableau method for temporal logic: An overview. Logique et Analyse, (110-111), 119-136.

[36] https://en.wikipedia.org/wiki/Computation_tree_logic

[37] Clarke, E. M., Grumberg, O., Peled, D. (1999). Model checking. MIT press.

[38] Miecicki, J., Czejdo, B., Daszczuk, W. B. (2017). Model checking in the COSMA environment as a support for the design of pipelined processing. arXiv preprint arXiv:1705.04728.

[39] http://projects.laas.fr/tina/

[40] https://www2.informatik.hu-berlin.de/ starke/ina.html

[41] S. Roch and P.H Starke P.H, Integrated Net Analyzer, User manual, 2002.

[42] https://en.wikipedia.org/wiki/List_of_model_checking_tools

[43] http://spinroot.com/spin/whatispin.html

[44] http://sal.csl.sri.com/

[45] http://www.doc.ic.ac.uk/ jnm/book/

[46] http://www.kenmcmil.com/smv.html

[47] https://fldit-www.cs.uni-dortmund.de/p̃eter/ExpNeu/Welcome.html

[48] http://lamport.azurewebsites.net/tla/tla-intro.html