Translating Statecharts to behaviourally equivalent Timed Petri Nets

Matteo Guastella[1]

University of Antwerp, Belgium

Abstract

Nowadays, when we model complex and real-time systems is very important to pay attention not only on the system behaviour, but also to the fact that the system must respect some properties, like for example, reliability and safety. This kind of properties can be verified with a Petri Net, but building a complex system using it is very difficult, because the complexity of the resulting model can explode. Hence we want to build a system in an "easy" way watching at his behaviour, using Statecharts, and then we want to convert it "automatically" in a model that is much more effective in analysis of properties, like Petri Net. For this purpose this paper will give an explanation of how to express Statecharts by means of Timed Petri Nets. We will show how to make the transformation using a rule based approach implemented in AToMPM and then we will discuss on the correctness and the limitations of the transformation.

Keywords: Model Transformation, Statechart, Timed Petri Net, TINA toolbox, AToMPM

1. Introduction

Model complex systems can be very hard, for this reason a behavioural oriented approach is preferred in most of the cases. But this approach cannot be enough, because we are only handling the functional requirements of the system, ignoring completely the non functional. Indeed when we talk about complex systems, usually we talk about system that interact with an external environment, that requires an answer in a certain amount of time. Hence

[1]email: matteo.guastella@student.uantwerpen.be

aspects such as reliability and safety becomes fundamental in the modelling process of systems such as those described above.

If for example, we want to model a driverless controller of a train, we want to know if the controller can goes in deadlock. For this reason the Statechart formalism is not enough, but we need another kind that is specialized in analysis of properties, like Petri Nets.

However, for complex systems it's very hard to be able to model a Petri Net, for this reason it would be very beneficial an automatic transformation between Statecharts and Petri Nets. In this way we can keep the facility of modelling, deriving from Statecharts and at the same time we can improve the analysis of the system, using the derived Petri Nets. Unfortunately we can't use the "canonical" Petri Nets, because the Statecharts have a richer semantic. Hence some elements typical of Statecharts are not expressible by Petri Nets. First of all, the concept of time, for this reason we will use the Timed Petri Nets formalism as specified in [1]. For other elements we will just define the limitation of our transformation.

Once the target model is created, we need to verify if it is behavioural equivalent to the corresponding Statechart. A proposed method by [1], is to produce a number of sequence diagrams based on the possible actions performed by the statechart, and verify if the reachability graph of the petri net is consistent with them. Another way to verify the "goodness" of the transformation, is to create a test suite with a series of relevant statechart, and the equivalent timed petri nets produced manually. Then we can automatically derive the statecharts in the test suite with the transformation and see if the Timed Petri Nets from the test suite and those automatically generate are the same, or are at least behavioural equivalent. The latter approach is indeed the one that we have followed. In particular we will assess the equivalence between the models watching at the reachability graphs.

The paper is structured as follow: In section 2 we will talk about the semantic of Statechart described in [2] and [3], we will also argue about different aspects of the semantic, to implement during the transformation; in section 3 we will talk about Timed Petri Nets implemented in AToMPM, its semantic and the connection with TINA's TPNs. [4]; in section 4 we will describe concretely the rule based transformation using AToMPM; in section 5 we will describe the step necessary to convert AToMPM petri nets to TINA petri nets; finally in section 6 we will discuss about the work done and the improvements that can be made.

2. Semantic of Statecharts

In this section we will shortly describe the semantic of the Statecharts, taking inspiration from [2] and [3]. We didn't follow a specific implementation of StateCharts, but we just considered the different options offered with their pros and cons, and we chose the one that looks better for our purposes. In most of the cases we tried to implement more than one option, so the transformation can suite different needs.

A Statechart describes the interactive behaviour of a system. The active configuration represents the current state of the system, in other words represent a snapshot of the system in a certain point in time. Instead the behaviour of the statechart is represented by a sequence of active configurations which follow one another in time, we can see this like the semantic of the Statecharts. It should be noted, that it doesn't exist a formal definition of statecharts semantic, unlike for example Petri Nets. Hence another reason for this transformation is that it makes the semantics of Statecharts explicit in term of Petri Nets formalism.

Another remark to do is that the Statecharts are typically deterministic. So when we will convert the model, the resulting Petri Net will have a "linear" reachability graph (in the following sections we will clarify this concept).

As we can see in figure 1 the statecharts have three types of states: basic-state, OR-state, AND-state. The OR-states have subcomponents but only one can be active at the same time, the AND-states have a number of orthogonal components, that are executed in parallel. These components share the events. So if an event is generated in an orthogonal component is broadcasted to the others orthogonal components possibly activating other transitions. This can produce a chain effect that we should handle in our transformation.

As we can see in figure 2 a transition can have the following label: **"m[c]/a"**, where **m** represent the message that trigger the transition, **c** the condition that control the execution of the transition, and finally **a** represent an action that is performed after the transition is triggered. All of these parts are optional. We can also have a time-out trigger message, that has a parameter t that represent the time that the transition has to wait before the action can occur. Certain actions, in addition to being performed when a transition is activated, can be performed at the entrance or at the exit of a state, in this case we talk about *entry action* and *exit action*.

The set of transition and action of which we talked about, represent the reactive behaviour of the system that reply to external(events) or tempo-

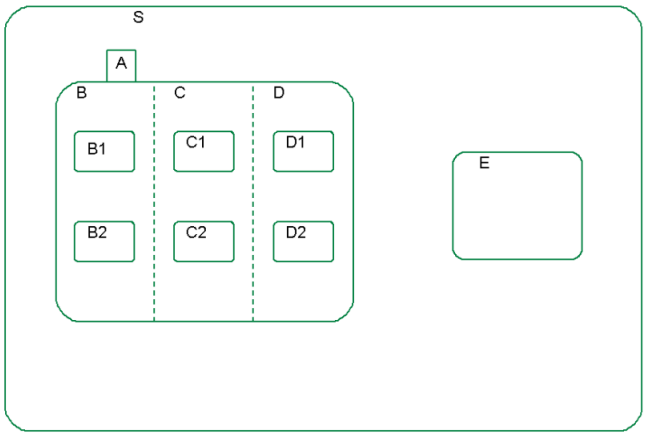

Figure 1: Hierarchy of States

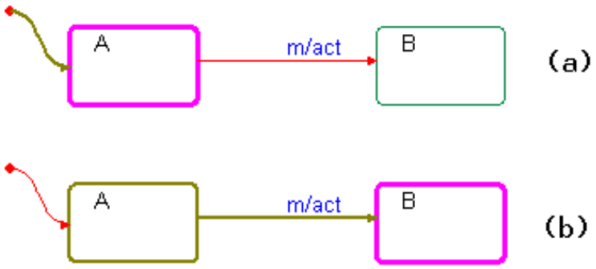

Figure 2: Transition

ral(planned actions) stimuli.

Another important element of Statecharts is the history connector. This connector is used for take in memory the active configuration of a component when the system goes out of it and then returns in it, an example is showed in figure 3. To the history connector we can associate two kind of semantic: "shallow history" and "deep history". The first take in memory the active configuration of the state to only one level of depth, instead the second goes inside the sub-states recursively until it finds only basic states, in this way can memorize the entire active configuration of a composite state.

The last important aspect of which we need to talk is how the Statechart formalism represent the time. Statemate implementation [3], defines two type of model of time:

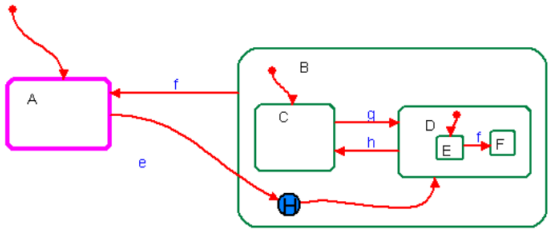

Figure 3: History

- **synchronous**, assumes that the system executes a single step every time unit, reacting to all the external changes that occur between the two time units;
- **asynchronous**(greedy), assumes that the system reacts whenever an external change occurs, allowing for several external changes to occur simultaneously and, most importantly, allowing several steps to take place within a single point in time.

3. Timed Petri Net formalism in AToMPM

In this section we will show the Timed Petri Net formalism created using AToMPM. In which we defined the elements that we need for executing the transformation described in the next section. Always keeping in mind what can be expressed in TINA, because at the end of the transformation we should be able to convert the model to a valid syntax net for TINA.

Important elements for our transformation are timed transitions and inhibitor arcs both represented in the following abstract syntax. As we can see in figure 4 we have two classes: Transition and Place. The Place class has two attributes:

1. **pname**, name of the place;
2. **tokens**, number of tokens in the place;

The Transition class has four attributes:

1. **tname**, name of the transition;
2. **interval_min**, lower limit of the interval, accept int values;
3. **interval_max**, uppre limit of the interval, accept int values;

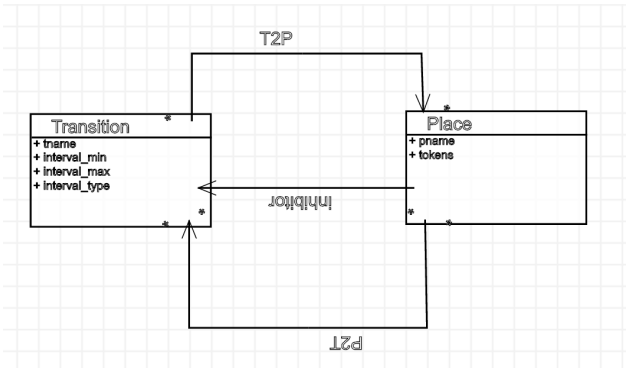

Figure 4: Abstract Syntax Timed Petri Nets

4. **interval_type**, boundaries of the interval. We can have four kind of interval, depending on the needs (open, close intervals), but usually we will use close intervals with an unique value. Because in Statecharts the interval is defined and has a single value and not a range. However is useful to create a general formalism for reusing it;

For the transition class we have four constraint, the first two ensure that the syntax of interval type is correct. The third prevent from negative value for min e max and the latter ensure that the min is less than or equal to the max value. Finally we have three kinds of arcs: T2P, P2T, Inhibitor. The first two are the usual arcs between transitions the last one activate the transition only if the place connected to it is empty.

I also modified the Concrete Visual Syntax accordingly to the Abstract Syntax of the formalism. As we can see in figure 5 is added to the canonical syntax an interval above the transition, that change its aspects based on the values inserted using the mapper. And also another kind of arcs representing the inhibitor arc.

4. Transformation

In this section we will introduce the set of rules that are used to transform Statecharts in Timed Petri Nets. This section is divided in paragraph, in each of which we will show a pattern of a Statechart and the corresponding rules to apply for obtaining an equivalent TPN. At the end of this section we will show the chain of rules applied in every transformation.

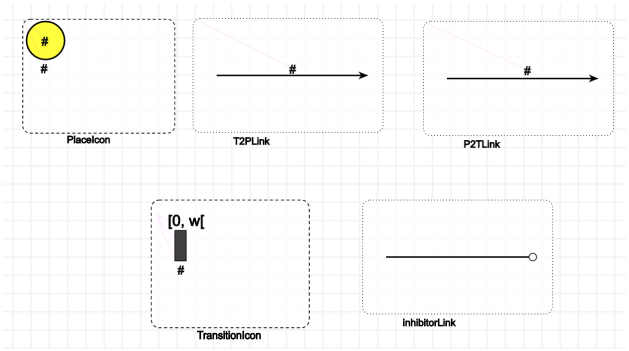

Figure 5: Concrete Visual Syntax

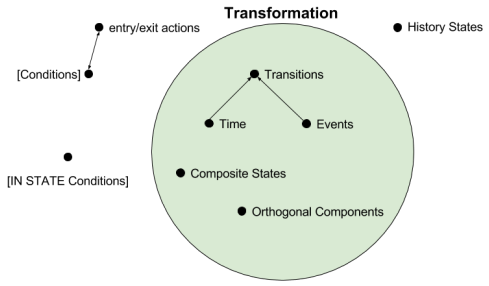

Figure 6: Boundaries

But first is fundamental to define the boundaries of the transformation. In other words we need to define which elements of the Statechart formalism are involved in the transformation and which are left out of it. As we can see in figure 6 the transformation will include, transitions triggered by events and time intervals, OR-states, AND-states. We left out the entry/exit action and the transitions triggered by conditions, because they use variables that should be stored in memory. The problem is that the Petri Nets formalism doesn't support this concept as so we should use other variants of Petri Nets like Colored Petri Nets. We also left out the history state, but is easily implementable using the pattern provide in [1].

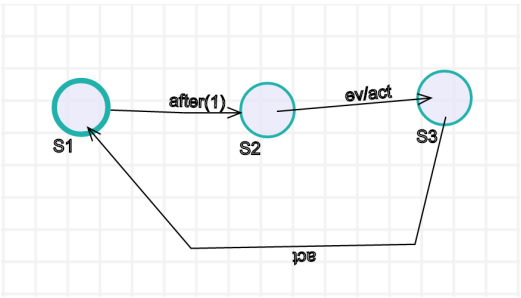

Figure 7: Transitions example

Initialization MacroStep. This paragraph describe the initialization steps involved at the beginning of the transformation:

1. **Solve conflicts between transitions:**, this rule is used in case of a possible non deterministic behaviour of the model, due to the presence of an internal and an external transition with the same name. In this case, following the statemate approach described in [3], we will delete the internal transition giving higher priority to the external transition. An example of this kinds of conflicts can be found in figure 10;
2. **flatten rename**, is used for flattening the Statechart. It rename the states recursively until a simple state is found using the following pattern: "parentstate.state";
3. **init**, create a place for each simple state in the statechart.;
4. **mark initial place**, search the initial place of the statecharts and mark the corresponding place with a token.

Transition Arc Between Simple States. As we can see in figure 7 a transition between two states can be simple, without triggers, or can be triggered by an event or by a time interval. For this reason we need to distinguish three different rules for the transformation.

1. **Triggerless:**
   This transition is the simplest, because we can just create a transition between the places involved.
2. **Call event:**
   As we can see in the figure 8, it search a pair of states that have a transition triggered by an event. It creates the transition "act" corresponding to the action performed by the transition and another place

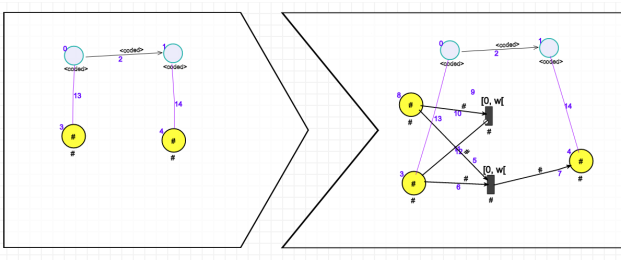

Figure 8: Event transition rule

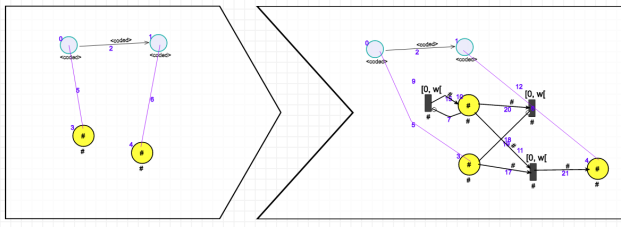

Figure 9: Timed transition rule

representing the event "e", if the place is marked it means that the event is occurred. For this reason we have to introduce another transition "abort", because the event can occur but the system could not be in the corresponding state, so we need to discard it. Every time the system is in the right state and an event occur the corresponding transition must fire. So we can just connect to the abort transition an inhibitor arc from the state place, in this way only if it is empty (the system is not in this state) the abort transition can fire.

3. **Time event:**
   As we can see in figure 9 the rule is very similar to the previous, but in this case we need to wait a certain amount of time before firing the transition. For this reason the rule creates a transition "T" that after "d" time fires and marks with a token the event place "e". We also need an inhibitor arc between the event place and the transition, because the timed transition can fire only if the event place "e" is empty (i.e. the event is not occurred).

The patterns expressed above are valid also for transitions between states at every level of hierarchy. For this reason starting from here when we are talking about a transition we are implicitly referring to the event transition.

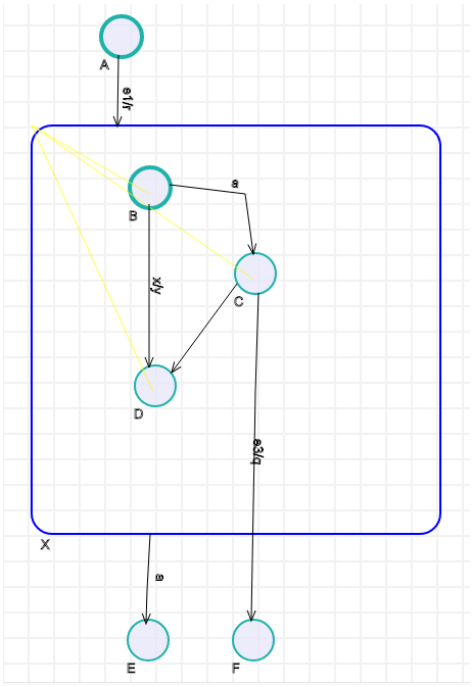

Figure 10: Transition from and to composite states

Variation of this rule can be easily derived.

In figure 10 we can find a simple example with different kinds of transitions that now we are going to explain, and convert.

Transition Arc from a Simple State to a Composite State. In this case we can distinguish between other two cases, the first in which the transition enters a sequential composite state, and the second in which the transition enters a concurrent composite state. Both these rules follows a two steps approach:

1. **Initialization**, the transition is initialized but is only connected to the simple state(this rule is in common).
2. **Connection**, if the transition enter a sequential state the transition "act" is connected to the initial state of the composite state. Otherwise is connected to each orthogonal component and precisely to their initial states.

The rules involved are showed in figure 11

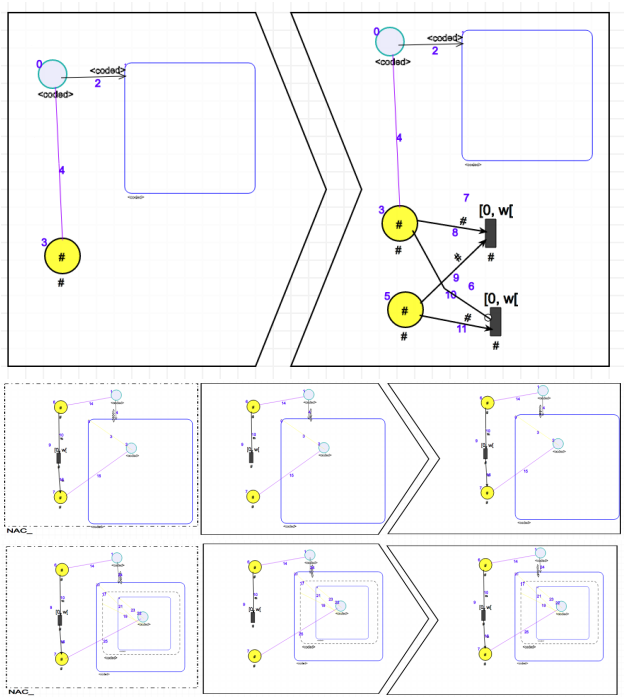

Figure 11: From simple to composite state

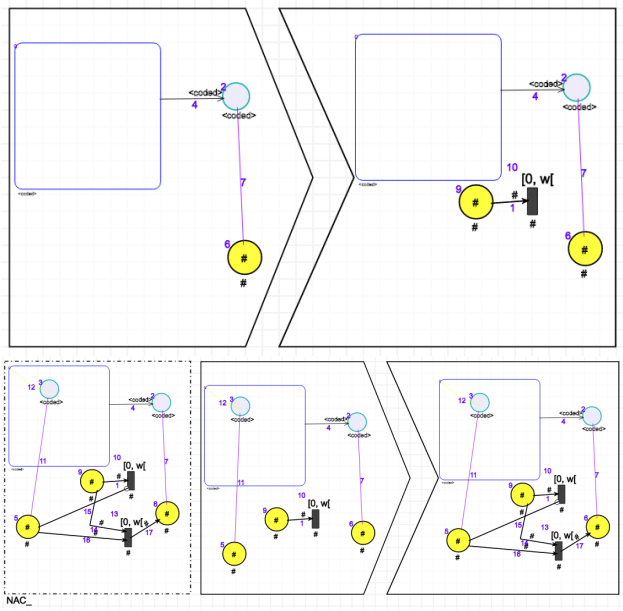

Figure 12: From composite state to simple state

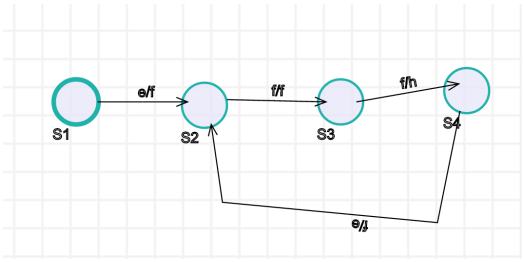

Figure 13: Interesting example about time and events

Transition Arc from a Composite States to a Simple State. In this case we can have two different types of transitions outgoing from the composite state.

1. **A transition from a composite state**, in this case we need to connect every sub-state in the composite state to a transition "act" that is triggered by the same event. In this way we construct a XOR between state, because from every state we can trigger the transition and go out of the composite state, but only one state at the same time can be occupied.
2. **An exit transition from one sub-state of the composite state, to a new target state**. In this case is just a normal transition between simple states, because we have already flattened the model.

In figure 12 are showed the rules involved. We can also have transitions between composite states, the rules involved are very similar to those just explained, so we are not going to explain them.

Model of Time. As we can see in figure 13, we can have transitions that generate an events, and the event just generated can been immediately available for other transitions, in this case the next one. So the event is generated and consumed in the same time step. As we said before we can have two kind of approaches in this case: synchronous and asynchronous. For a better solution we have chosen to implement both.

The asynchronous is just the one that we have already implemented. For the synchronous I created an alternative schedule (*T_schedule_syn*). The idea behind is that only an "action" transition can fire in the same time step. So I created a sort of fair pattern that allow only one transition per time step to fire. In this way is not possible to fire two transition at the "same" time.

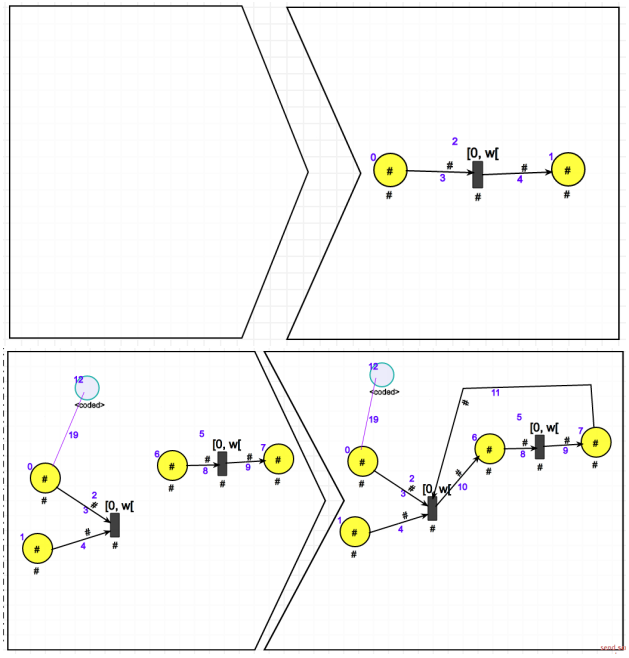

Figure 14: Synchronous model of time

Broadcast of Events. As I already explained between orthogonal components the events are sent in broadcasting. So we need a transformation that preserve the events and at the same time gives the same opportunity to fire by each "act" transition. So as we can see in figure 15 the transformation use the following steps:

1. create a double connection between transitions and events, in this way the events are never consumed;
2. create the transition "fair";
3. create for every "act" transition an in_fair place and an out_fair place, and connect to them the "act" transition and the "abort" transition. Connect also the abort transition, because if the state is not current the transition should fire anyway, otherwise the system will go in deadlock.
4. connect the event place to the fair transition, because at every time step all the events should be discarded.

Generate the Environment. A statecharts is a reactive system that change its internal state and produce output in answer to external events that are given

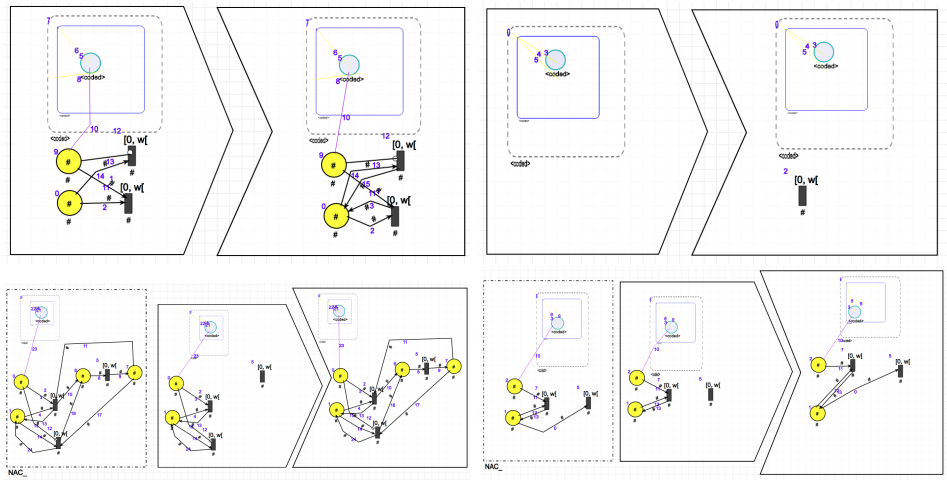

Figure 15: Asynchronous model of time

from an environment. After our transformation we have modeled the system and now we miss to model the environment. Otherwise we cannot simulate the system and we cannot produce the reachability graph. A schematic description of what we want to obtain is showed in figure 16. Fortunately the environment is just a non deterministic event generator. A perfect formalism for this purpose is a non deterministic Petri Net.

Hence the last step of the transformation is to create this event generator and connect it to all the events in the model. The main characteristic of this generator is that is a big choice pattern between every events, so at every step of the simulation we can randomly generate one of the events if it not already present, figure 17 show the rule involved in this transformation.

Schedule. Finally we can execute the rules described above using the schedule showed in figure 18, the order of the macro steps involved in this transformation are the following:

1. Initialization
2. Handle transitions to composite states
3. Handle transitions from composite states
4. Handle transitions between composite states
5. Handle transitions between simple states
6. Handle events and broadcasting inside Orthogonal Components
7. Events Generator

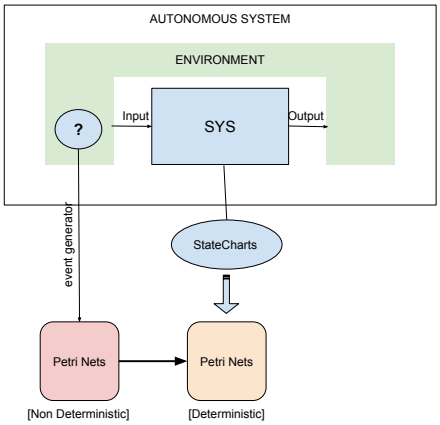

Figure 16: Autonomous system

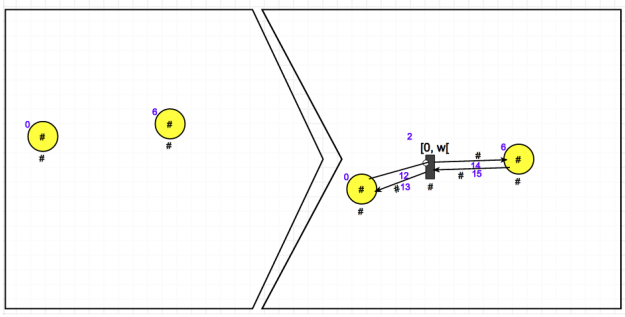

Figure 17: Event generator

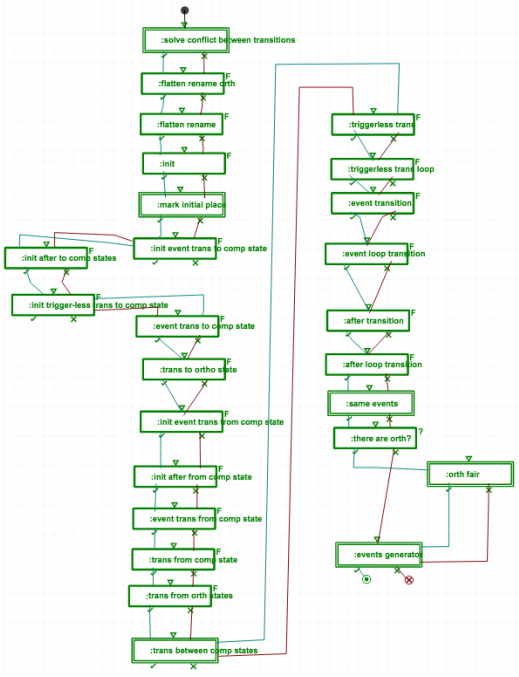

Figure 18: T_schedule_asyn.model transformation schedule

In point 5 in the synchronous approach is followed by the application of the rule explained above before the event generator.

5. From AToMPM to TINA toolbox

If we apply the schedule to a valid Statecharts we construct the corresponding Timed Petri Net in AToMPM. The next goal is to derive from the Net obtained a TINA's valid Net, that we can import and analyze. With this goal in mind we need to follow this approach:

1. export the model in metaDepth using the corresponding toolbar;
2. generate the file *exported.tpn* using EGL language;
3. import in TINA toolbox and analyze.

At this point we need to assess the goodness of the transformation, the easiest way to do it is just watch at the net and see if the patterns explained in the rules are present. A better approach is to construct the reachability graph and see if the behavior and the typical characteristics of Statecharts emerge in it.

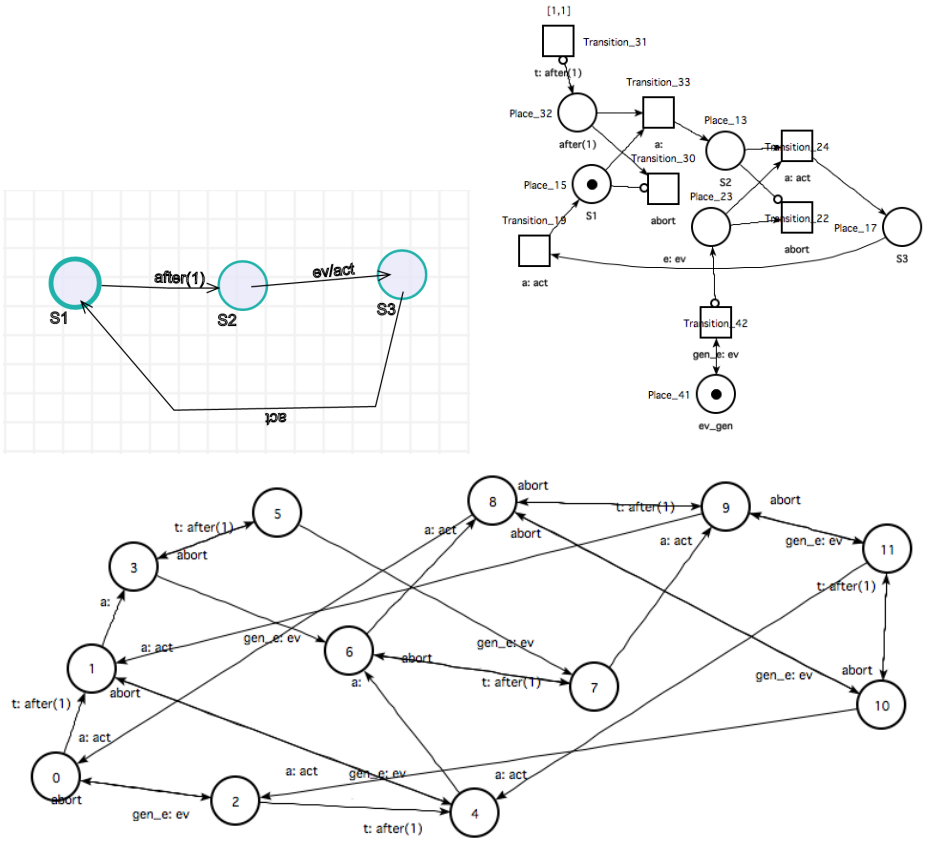

Figure 19: Example of transformation

Now that all the steps are in order we can see an example of transformation. In figure 19 we can see a simple example and its corresponding transformation imported in TINA. We can see some typical patterns like timed transitions, event generator and so on. But we want also to analyze the net and build the reachability graph that is showed in figure 19. As we can see the reachability graph is linear if we consider the transitions triggered by the actions "a:" these actions indeed represent the deterministic behavior typical of Statecharts. Instead the "environment" transitions, like "abort", "after", and "ev_gen" represent the non deterministic behavior of the autonomous system.

6. Conclusion

In this project we successfully extended and implemented the approach showed in [1]. Creating an automatic transformation from Statecharts to Timed Petri Nets. This transformation bring with it some advantages:

- improve the analysis of properties of systems modeled using Statecharts;
- define a rigorous semantic for Statecharts by means of the semantic of Petri Nets (reachability graph).

Future work should concentrate its attention first, on improving the transformation: for example generalizing the rules using abstract state and trying to reduce the number of rules, or trying to improve the performance of the transformation, because with big statecharts strongly connected the transformation can take a while before terminate. Second extending the boundaries defined previously, maybe trying to transform to a richer Petri Nets formalism.

References

[1] Y. Hammal, A Formal Semantics of UML StateCharts by Means of Timed Petri Nets, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 38–52. doi:10.1007/11562436_5.

[2] D. Harel, H. Kugler, The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML), Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 325–354. doi:10.1007/978-3-540-27863-4_19.

[3] D. Harel, A. Naamad, The statemate semantics of statecharts, ACM Trans. Softw. Eng. Methodol. 5 (4) (1996) 293–333. doi:10.1145/235321.235322.
URL http://doi.acm.org/10.1145/235321.235322

[4] Tina (time petri net analyzer).
URL http://projects.laas.fr/tina/index.php