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 to the behaviour of the system, but also to the fact that the system must respect some properties, like 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 the 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. Based on the semantic of these two formalisms we will show how to make the transformation using a step by step approach, with examples.

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

1. Introduction

Model complex systems can be very hard, for this reason often it's used an approach that concentrate its effort only on the behaviour of the system, in this way we handle only the functional requirements of the system. This approach is perfectly catch by the Statecharts, with which we can simulate the deterministic behaviour of a system. But something is missing, because when we talk about complex systems, usually we talk about system that interact

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

with the external environment, and that must answer in a certain amount of time. Hence aspects such as reliability and safety becomes fundamental in the modelling process of this kind of systems.

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 others, that are 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. Because, 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 the Petri Nets. For example the possibility to create timed events, for this reason we will use the Timed Petri Nets or as specified in [1], Interval Timed Petri Nets. As you can tell from the name, this formalism introduce the concept of time in the petri nets, so we can have timed transitions.

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. We can also check the temporal requirement because the arcs of the reachability graph are marked with the interval time of every transition. 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 paper is structured as follow: In the section 2 we will talk about the semantic of Statechart described in [2] and implemented in the RHAPSODY tool; in the section 3 we will talk about the semantic of Timed Petri Nets throw the description of TINA toolbox [3]; in the section 4 we will describe concretely the transformation, and for every element in the Statecharts we will describe a specific transformation rule; Finally in the section 5 we will talk about the steps to do for implementing this transformation.

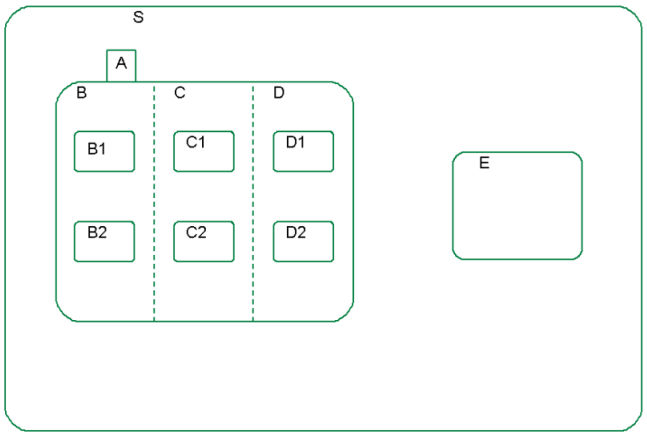

Figure 1: Hierarchy of States

2. Semantic of StateCharts

In this section we will shortly describe the semantinc of the Statecharts, taking inspiration from [2], more precisely we will describe an Object-Oriented version of Statechart.

A Statechart describe the behaviour of the system. The active configuration represent 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 the statecharts semantic, unlike for example the Petri Nets. 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.

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.

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. A transition can be triggered by an event or by a

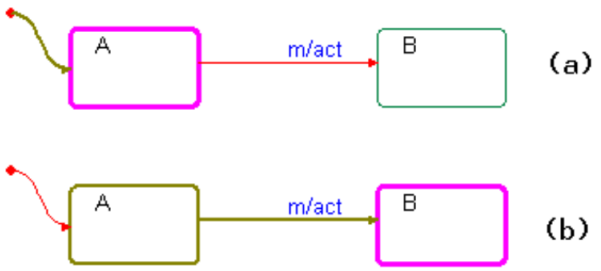

Figure 2: Transition

triggered operation, the difference between these messages is that the first is asynchronous and the second is synchronous. 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 action can occur. Certain actions, in addition to being performed when a transition is activated, can be performed at 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 temporal(planned actions) stimuli.

A last 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 "shallow history" is not supported by the RAPHSODY tool, but fortunately it is supported by the formalism of the Statecharts in AToMPM. That is the one that I will use for my project.

3. Semantic of Timed Petri Nets throw TINA toolbox

In this section we will follow a practical approach, indeed we will try to define the elements of the Timed Petri Net that we need for executing the transformation described in the next section. In particular, we will describe the semantic of TPN throw some examples produced with the TINA toolbox.

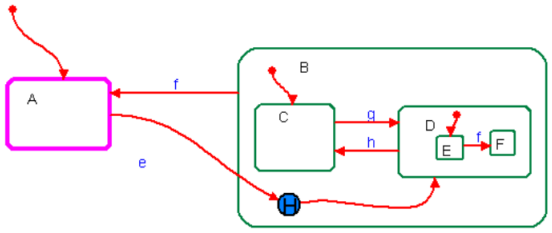

Figure 3: History

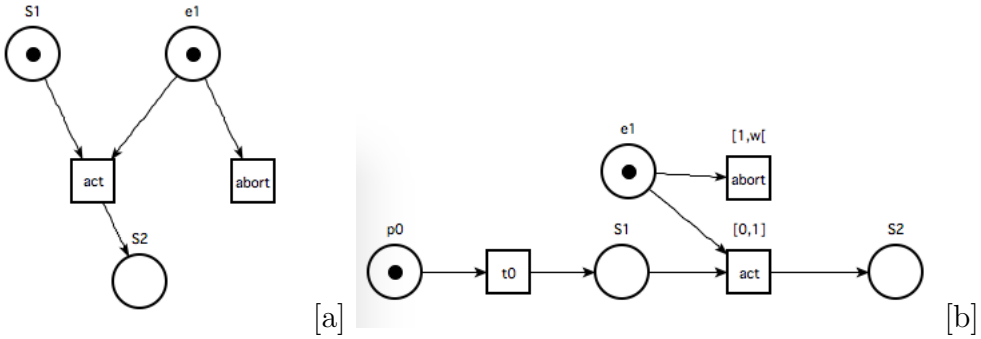

Figure 4: Some examples

Obviously we know that the real semantic of a Petri Net is represented by his reachability graph, but we want to follow a more practical approach.

TINA toolbox, allow to model normal Petri Nets, so we can create, place, token, transition and arcs (simple, inhibitors). In addition allow to create Timed Petri Nets, so for every transition we can define an interval of time in which the transition is active. In figure 4 we can see a normal Petri Net and a Timed Petri Net, created with TINA. Unfortunately in [1], as we can se in the next session, the concept of interval is applied to the outgoing and ingoing edges in the transitions, hence from a transition we can have most active arcs at different instants of time. We can handle this problem by creating a transition for every arc.

Another useful characteristic of TINA Petri Nets is that we can define also priority between transitions, like in the figure 5. As we can see in the next section this aspect will be very useful.

The elements described above, like interval transition, priority are not defined in the formalism of Petri Net implemented in AToPM. An additional

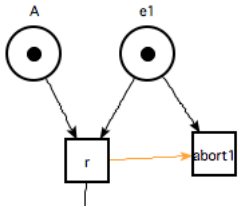

Figure 5: Some examples

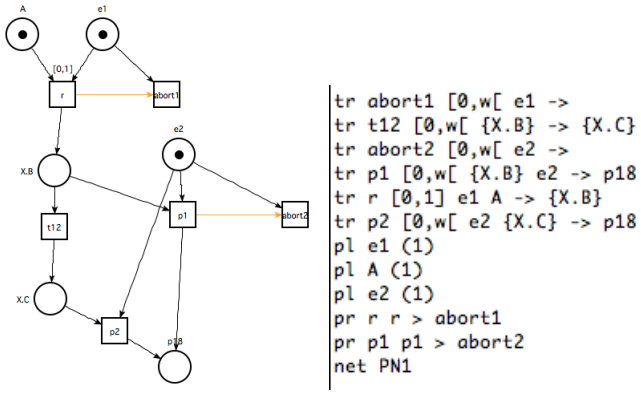

Figure 6: Textual representation of a net

task of my work will be to extend the formalism (modifying the abstract syntax), to adapt it to the needs of the transformation.

Another feature of TINA is that it allow to import and export the Petri Net in a simple and intuitive textual format under the format ".tpn", in the figure 6 we can see a simple example. This feature is very important, for the purpose of creating an "automatic" transformation" between the models. Because in this way we can create a tpn file directly from the model generated by AToPM, realistically using EGL, obtaining a format ready to be analysed.

A last feature to consider, is that TINA obviously allow us to do analysis on the PN, like boundedness, reachability, liveness. As we have already said this kind of analysis are very useful for the final purpose of this project.

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 rule

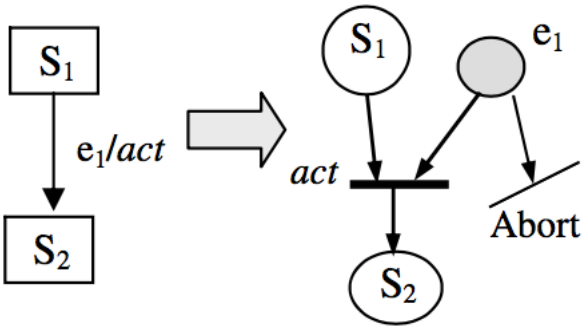

Figure 7: Call transition

to apply for obtaining an equivalent TPN. This approach is very useful in order to apply a rule-based transformation to the model using AToMPM.

Transition Arc Between Simple States. A transition between two states can be triggered by an event, or can be triggered after a certain amount of time, for this reason we need to distinguish two different rules for the transformation.

1. **Call event:**
   As we can see in the figure 7, it creates a place for every state in the SC, and it creates also the transition "arcs" that correspond to the action performed by the transition. Additionally we need to create another place for 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 the event. Unfortunately this transition generates a problem, because we can have a situation in which the transition "act" can fire, but the transition "abort" fires first. We need to avoid this kind of behaviour because every time the system is in the right state and an event occur the corresponding transition must fire. Hence to solve this problem we need to introduce priority between arcs, as we have already seen this is very easy using TINA.
2. **Time event:**
   As we can see in the figure 8 the rule is very similar to the previous example, but in this case we need to wait a certain amount of time before the transition can fire. For this reason we can create a transition

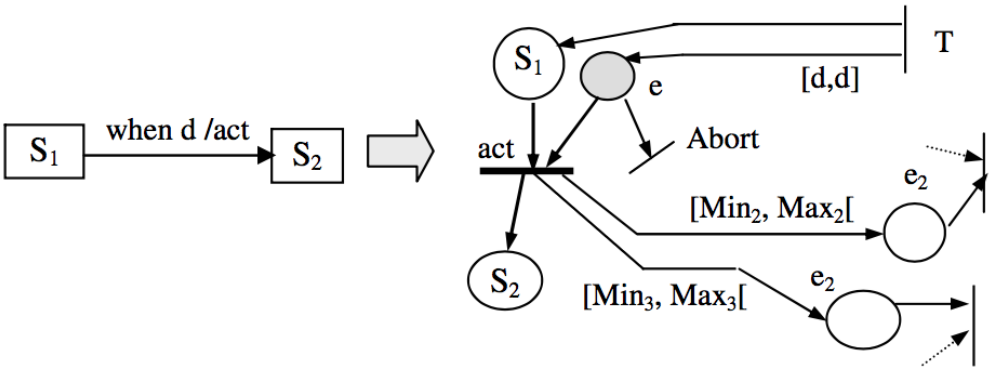

Figure 8: Timed transition

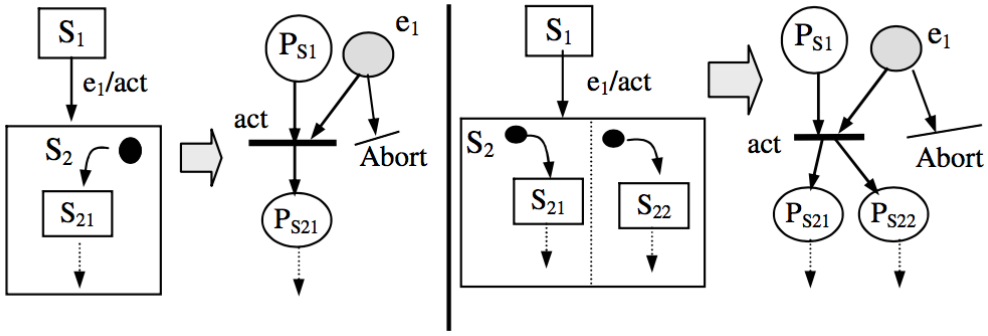

Figure 9: Sequential and Concurrent composite states

"T" that marks with a token "S1" immediately, and after "d" time, in this case, fires another time and marks with another token "e". Seeing that in TINA we can't apply time intervals to the arcs, but only to the transitions, it will be slightly different the final implementation of the rule.

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. The first that we can see in figure 9 is practically identical to the case of a transition between two simple states. The second is

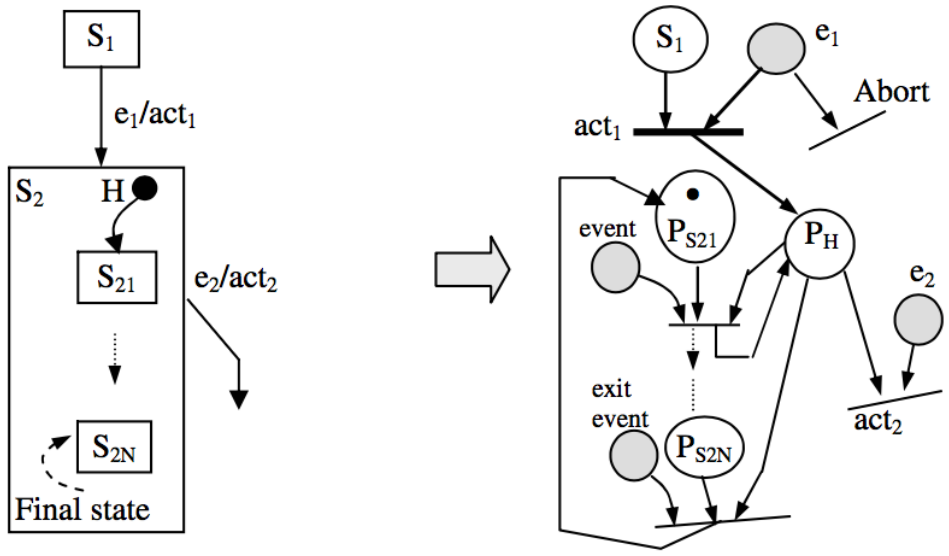

Figure 10: Sequential composite state with history

easy too, because we just have to connect the "act" transition to every place that is initial in his corresponding orthogonal component.

Transition Arc from a Simple State to a Composite State with History. When we have a composite state with a history state, the behaviour should be the following. If the state is exited the history state take in memory the state of the system. When the state is entered again the memorized state should be restored. In this paragraph we distinguish two different composite state: sequential composite state and concurrent composite state.

1. **Sequential composite state**, as we can see in figure 10, we need a special place "PH" that handles the outgoing and ingoing transitions in the composite state. Hence if an outgoing transition from "S2" is active, "PH" is emptied, and only when an ingoing transition in "S2" will fire, the place "PH" will be marked again. So the current state of the system is given only by the token that goes between the internal states of the composite state. Indeed at the beginning of the execution of the system the initial state of the composite state will be marked with a token, and the token will never goes out the scope of the composite state. In this way when "PH" will be remarked the state it

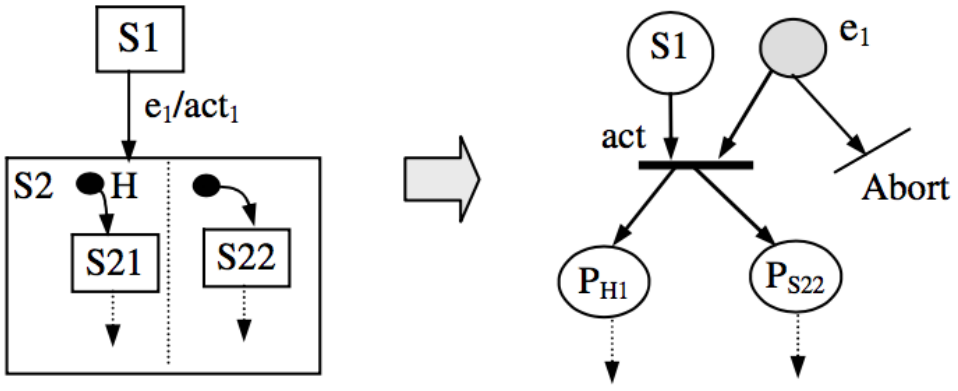

Figure 11: Concurrent composite state with history

will remember its last active configuration.

2. **Concurrent composite state**, the only important thing to remember is that if we have an orthogonal component with the history state, we need to connect the ingoing transition to the place that represent the history state and not to the initial state place, for the reasons that we have explained previously. An example is showed in the figure 11

Transition Arc from a Composite States. As it is clearly visible in the figure 12, because in the Petri Nets there isn't the concept of hierarchy between the elements, we need to call every place with an unique name and preferably maintain a hierarchy between the names. In this way we can easily discover the corresponding state of the statechart.

In this case we can have three different types of transitions outgoing from the composite state.

1. A trigger less transition outgoing to G, in this case we need just to connect the place representing the final substate of X, in this case D with the place G.
2. High-level transition outgoing from X to E, in this case we have to connect every sub-place of X with E. We need also an event place, because the transition is triggered by the event e2. In the figure 12 we can see dashed lines, that represent a XOR, hence only a place can be active at the same time in the transition. We can transform this abstract notation in a concrete one creating a transition p for every place.

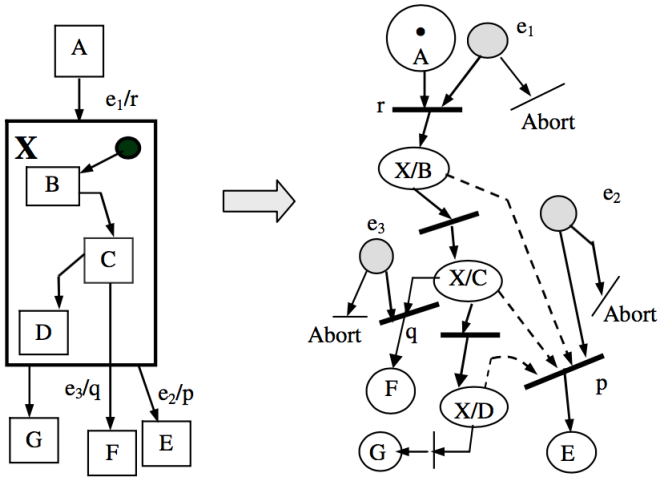

Figure 12: Different type of arcs from a composite state

3. An exit arc from one substate to a new target state, in this case we can just connect the sub-place C with the F place.

5. Conclusion

The goal to the project is to successfully transform a Statechart in a Timed Petri Net and finally prove the correctness of the transformation. So the step to follow are listed below:

- In AToPM extend the abstract syntax of the Petri Nets(already present), for representing the Timed Petri Nets used by TINA.
- Implement a rule-base transformation for converting Statecharts in behavioural equivalent Timed Petri Net.
- Export the model in a text format readable by TINA.
- Finally I have to verify if the models produced are equivalent, using a test suite with a certain number of statechart and the corresponding manual translation in Time Petri Net.

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

[4] Time petri nets: Theory, tools and applications.
URL http://www2.informatik.hu-berlin.de/~popova/tutorial.html