On the Use of Graph Transformation in the Modeling and Verification of Dynamic Behavior in UML Models

Elhillali Kerkouche
Department of Computer Science, University of Oum El Bouaghi, Algeria
elhillalik@yahoo.fr

Allaoua Chaoui
MISC Laboratory, Department of Computer Science, University of Constantine, Algeria
a_chaoui2001@yahoo.com

El Bay Bourennane and Ouassila Labbani
University of Bourgogne, LE2I Laboratoire, Dijon, France
{ebourenn, Ouassila.Labbani}@u-bourgogne.fr

Abstract— **The use of the UML specification language for modelling dynamic behaviors of systems is very widespread. UML Statecharts and Collaboration diagrams are widely used to model dynamic behaviors of systems. However, the lack of firm semantics for the UML modeling notations makes the detection of behavioral inconsistencies difficult in the initial phases of development. The use of formal methods makes such error detection possible but the learning cost is high. Integrating UML with a suitable formal notation is a promising approach that makes UML more precise and amenable to rigorous analysis. In this paper, we present the benefits of a similar approach that is the integration of UML Statechart and Collaboration diagrams and Colored Petri Nets models. The result is an automated approach and a tool environment that formally transforms dynamic behaviors of systems expressed using UML models into their equivalent Colored Petri Nets models for analysis purposes. To make the analysis easier, the obtained models are used to generate automatically their equivalent description in the input language of the INA Petri net analyzer. The approach is based on Graph Transformation and the Meta-Modeling tool ATOM^3 is used. The approach is illustrated through an example.**

Index Terms—**UML; CPN; INA Analyzer; Meta-Modeling; Graph Grammars; Graph Transformation; Models Transformation; AToM^3**

I. Introduction

The Unified Modeling Language (UML) [6] is considered nowadays as a standard modeling language in the software development process of systems based on the Object-Oriented Paradigm. It consists of many diagrams. Some diagrams are used to model the structure of a system while others are used to model the behavior of a system. UML Statecharts and Collaboration diagrams are widely used to model dynamic behavior of complex and concurrent systems in UML [20]. UML Statechart diagram models the lifetime (states life cycle) of an object in response to events, whereas a UML Collaboration diagram models the interaction between a set of objects through the messages (or events) that may be dispatched among them [6].

However, despite its success as being a unified and visual notation, UML diagrams still lack a precise formal semantics, which hinders the formal analysis and verification of system design. On the other hand, models established in many mathematical domains (such as Petri Nets, process algebras, transition systems, etc.) are precise and could be analyzed and verified by using various tools in these domains. Integrating UML with these models is a promising approach that makes UML more precise and amenable to rigorous analysis. Due to its understandability and abundant analysis techniques, Petri Nets (PNs) [28] are appropriate for modeling systems with concurrency. Colored Petri Nets (CPNs) [22] are a generalization of ordinary PNs, allowing convenient definition and manipulation of data values. CPNs also have a formal, mathematical representation with a well-defined syntax and semantics. Thus developing a tool support for modeling and analysis of complex concurrent systems is significant to modelers who use UML to model their systems. UML behavioral models are projected automatically into CPN models for analysis and verification to detect behavioral inconsistencies like deadlock, livelock, imperfect termination, etc. Then, the results of the formal analysis can be back-annotated to the UML models to hide the mathematics from modelers.

Building a modeling tool from the scratch is a prohibitive task. Meta-Modeling approach is useful to deal with this problem, as it allows (possibly in a graphical way) the modeling of the formalisms themselves [9]. A model of formalism should contain enough information to permit the automatic generation of a tool to check and build models subject to the described formalism's syntax. If this specification is done graphically, the time to develop a modeling tool can be

doi:10.4304/jsw.5.11.1279-1291

drastically reduced to a few hours. Since meta-model and model are graphs, further manipulations of the models can be described (modeled) graphically and formally as Graph Grammars [32], thus they become high-level modes, reducing the need for coding to a minimum. Some of these manipulations are model simulation or animation, model optimization, for example, to reduce its complexity, model transformation into another model (equivalent in behavior but expressed in a different formalism), and the generation of textual model representations for use by existing simulators or tools. These ideas presented above are implemented in ATOM^3 (A Tool for Multi-formalism and Meta-Modeling). It is developed at the Modeling, Simulation and Design Lab in the School of Computer Science of McGill University [2].

In this paper, we propose a Graph Transformation approach and tools for modeling and verification of dynamic behavior in UML models using CPN formalism. In order to get a more general transformation approach between UML and CPN, we research the transformation at the Meta-Model level. And for reaching an automatic and correct process, we use Graph Transformation Grammars and Systems to define and implement the transformation. Using our approach, the modelers specify the dynamics of a system by means of a set of Statechart diagrams and Collaboration diagram. Then the modelers transform automatically their behavioral specification into its equivalent single system-level CPN model. From this intermediate representation they can generate automatically the equivalent description in the input language of the INA analysis tool [21] for formal analysis and verification purposes.

With this end, we have defined simplified Meta-Models for UML Statechart diagram, Collaboration diagram and CPN formalism using AToM^3 tool. Than we have used this Meta-Modeling tool AToM^3 to automatically generate a visual modeling tool for each formalism according to its proposed meta-model. For the transformation process, we have defined three graph grammars. The first one converts the Statechart diagrams to Flat State Machine (FlatSM) [34] models. The second graph grammar transforms the obtained FlatSM models into CPN-like models and relates each other according to the message passing described in the Collaboration diagram. The resulting model of this step is a single system-level CPN model which is a form of Object Petri Nets called Object Net Models [33] (discussed in Section *IV*). The last graph grammar rebuilds the obtained CPN model in the input language of the INA analysis tool.

This paper is organized as follows. Section *II* outlines the major related work. In section *III*, we give an overview of the AToM^3 tool. Section *IV* summarizes briefly the pertinent concepts for formalizing UML using CPN which we require in our approach. In section *V*, we present our Graph Transformation approach. In section *VI*, we illustrate our framework with an example. Finally, section *VII* concludes the paper.

II. Related Works

In addition to AToM^3, there are several visual tools to describe formalisms using Meta-Modeling such as the Generic Modeling Environment (GME) [17], MetaEdit+ [23], KerMeta [26] and other tools from the Eclipse Generative Modeling Tools (GMT) project such as the Eclipse Modeling Framework (EMF) [13], the Graphical Editing Framework (GEF) [15] and the Graphical Modeling Framework (GMF) [18]. In most of these tools, model manipulations (transformation, simulation or code generation) have to be described textually, and user friendly support for visual analysis and testing is generally missing. In AToM^3, the user expresses such transformations by means of Graph Grammar models. Graph Grammars are a natural, declarative, and general way to express these manipulations.

There are also similar tools which manipulate models by means of Graph Grammars, such as PROGRES [30], GReAT [19], FUJABA [14], TIGER [35] and AGG [1]. However, none of these has its own Meta-Modeling layer. Some of them are complemented with support for Meta-Modeling (for example, The GReAT model transformation engine is combined with GME).

The combined use of Meta-Modeling and Graph Grammars taken in AToM^3 allow users not only to benefit from the advantages of both (Meta-Modeling and Graph Grammars) but also to model with Multi-Paradigm Modeling [7]. The AToM^3 tool has been proven to be very powerful, allowing the Meta-Modeling and the transformations of known formalisms.

In [8] the authors presented a transformation between Statecharts (without hierarchy) and Petri Nets. In [12], we have provided the INA Petri net tool [21] with a graphical environment. In [24], we have presented a formal framework (a tool) based on the combined use of Meta-Modeling and Graph Grammars for the specification and the analysis of complex software systems using G-Nets formalism. G-Nets [10] formalism is a kind of high level Petri Nets for the modular design of distributed systems. Our framework allows a developer to draw a G-Nets model and transform it into its equivalent Predicate/Transition Nets (PrT-nets) [16] model automatically. In order to perform the analysis using the PROD analyzer [29], our framework allows a developer to translate automatically each resulted PrT-Nets model into PROD's net description language.

In this paper we propose an extended version of our paper published in [25]. It consists of an approach for transforming UML Statechart and Collaboration diagrams to colored Petri nets models. More precisely, we have proposed an automated approach and a tool environment that formally transforms dynamic behaviors of systems expressed using UML models into their equivalent colored Petri Nets (CPN) models for analysis purposes.

III. Graph Grammar and AToM^3

AToM^3 [2] is a visual tool for Multi-formalism Modeling and Meta-Modeling. Being implemented in

Python [3], it is able to run without any change on all platforms for which an interpreter for Python is available.

By means of Meta-Modeling, we can describe or model the different kinds of formalisms needed in the specification and design of systems. The AToM^3 meta-layer allows a high-level description of models using Entity Relationship (ER) formalism or UML Class Diagram formalism extended with the ability to express constraints. Based on these descriptions, AToM^3 can automatically generate tools to manipulate (create and edit) models in the formalisms of interest [9].

AToM^3's capabilities are not restricted to these manipulations. AToM^3 also supports graph rewriting system, which uses Graph Grammar to visually guide the procedure of model transformation. Model transformation refers to the (automatic) process of converting, translating, or modifying a model of a given formalism into another model that might or might not be in the same formalism.

Graph Grammar [3] is a generalization of Chomsky grammar for graphs. It is a formalism in which the transformation of graph structures can be modeled and studied. The main idea of graph transformation is the rule-based modification of graphs as shown in Fig.1.

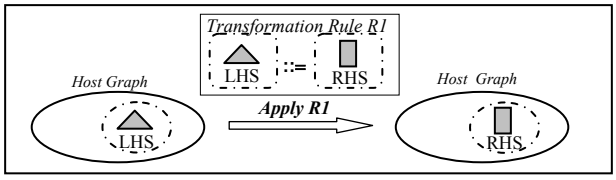

Figure 1. Rule-based Modification of Graphs.

Graph Grammars are composed of production rules, each having graphs in their left and right hand sides (LHS and RHS). Rules are compared with an input graph called host graph. If a matching is found between the LHS of a rule and a subgraph in the host graph, then the rule can be applied and the matching subgraph of the host graph is replaced by the RHS of the rule. Furthermore, rules may also have a condition that must be satisfied in order for the rule to be applied, as well as actions to be performed when the rule is executed. A graph rewriting system iteratively applies matching rules in the grammar to the host graph, until no more rules are applicable.

The use of a model in the form of a graph grammar of graph transformations has several advantages over embedding the computation in a lower-level, textual language [5]:

- Graph grammars are a natural, graphical, formal and high-level formalism.
- The theoretical foundations of graph rewriting systems can help in demonstrating the termination and correctness of the computation model.

IV. UML & CPN

The intuitive and graphical notations of Statechart diagrams provide the necessary expressive power for modeling dynamic aspects of complex and concurrent systems in the framework of UML [20]. However, the lack of a formal dynamic semantics for Statechart diagrams limits its capability to apply mathematical techniques analysis and verification. To overcome this limitation, several projects discuss transforming these diagrams into a formal model. Different types of PNs have been applied to this end. Despite new concepts (such as color, hierarchy, stochastic concepts) are fused into PNs, the essence of its syntax and its dynamic behavior properties never changes. In [11], Dong, et al. convert UML state machines into a type of Petri net called Hierarchical Predicate Transition Net (HPrTN). In [4], Merseguer et al. propose a formalization for a subset of UML Statecharts in terms of another type of Petri net call Generalized Stochastic Petri Net (GSPN). In [33] Saldhana et al. propose an approach to transform a set of Statechart diagrams into CPN model.

We note that these approaches have different strengths, but the latter approach provides a design strategy based on separation of concerns. This approach suggests that the Statechart diagrams are first converted into a Shlaer-Mellor object life cycle [34], which is a Flat state machine (FlatSM) model that only contains simple states and transitions. Since Statechart diagrams may contain hierarchical or nested states, effective conversion to CPNs requires that the nested states be "flattened". These FlatSM models are then converted to a form of Object Petri Nets (OPN) [27] called Object Net Models (ONM) [33]. Finally, the UML Collaboration diagram is used to connect these object Net models (ONMs) to derive a single system-level model for the system under study (see Fig.2.).

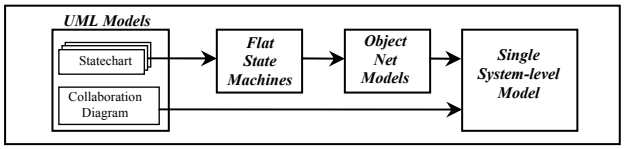

Figure 2. Architecture of Saldhana's approach [33].

The structure of ONM model consists of a lifetime behavior model (LM) and a token routing structure as shown in Fig.3. LM represents a CPN-like model that is derived from the Statechart of an object. Basically, the transformation from a Statechart to a CPN accomplished by the following mappings: a state is mapped to a place; a transition is mapped to a CPN transition. The concept of events is a key factor in defining the semantic of Statechart, the actions of creating, routing and dispatching of events determine its execution semantics. So, events are mapped to CPN tokens which define its data types.

The token routing structure defines three places (in-place (IP), out-place (OP) and event-dispatcher place (ED)) and four transitions (T1, T2, input transition arc (ITA) and output transition arc (OTA)). Since events are modeled by tokens, we refer to the tokens derived from Statecharts events by *event-tokens*. The IP place of the object holds the *event-tokens* that will be consumed by the object. Thus, arcs will connect the IP place to all transitions in the LM model that Statecharts transitions initiated by events. The ED place holds the *event-tokens*

that are generated by the object. So, for each transition in LM, if this transition generates a new *event-token*, there will be an arc targeting ED place. The generated *event-token* can have a type of either *external* or *internal*. If it is *internal*, it will be routed to IP place via transition T2. Otherwise, it will be routed to OP place via transition T1. The OP place of the object holds the *event-tokens* that will be routed to other objects. In order to define the communication between the objects, a special place (Internal Linking Place (ILP)) is used to route *event-tokens* between the ONM models. For precise detail see [33].

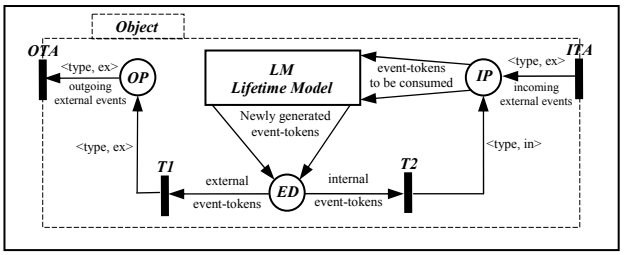

Figure 3. The structure of an Object Net Model [33].

We note that the transformation in this approach is performed manually. We propose in the next section our Graph Transformation approach to perform the transformation automatically. Since UML diagrams and CPN models are based on the graphical notations, there comes a possibility of depicting them by the common graph concepts, and with it the possibility of transforming UML models into CPN from the aspect of graph theory. Thereby, this is another reason for selecting PNs as the target formal model and considering the graph transformation as the foundation theory and highly automated mechanism for transformation.

V. Our Approach

In this section, we describe our automated approach and tools environment that formally transform dynamic behaviors of systems expressed using UML models into their equivalent CPN models for systems analysis using the INA Petri Nets analyzer. The approach is based on the combined use of Meta-Modeling and Graph Grammar. In this work, we assume that the behavior of the system is specified as a set of Statechart diagrams and that the Collaboration diagram is used to represent objects interaction. In order to derive the system-level CPN model from this behavioral specification, we have automated the approach proposed by Saldhana et al. [33]. To make the analysis easier, we have also automated the generation of the equivalent description of the obtained single CPN model in the input language of the INA analyzer (see Fig.4.).

Our approach consists of a process with two steps:

The first step consists of Meta-Modeling used UML diagrams and formalisms. More precisely, we have redefined meta-models for a basic category of UML Statechart and Collaboration diagram using the Meta-Modeling tool AToM^3. Likewise, we have also defined meta-models for FlatSM formalism and ONM formalism. Then, we have used AToM^3 tool to automatically generate a visual modeling tool for each of them according to their proposed Meta-Models.

The second step is to define the models transformation. In order to reach an automatic and correct process of transformation, we have proposed to use Graph Transformation Grammars and Systems to define and implement the transformation. So, we have defined three Graph Grammars:

1st GG: converts the Statechart diagrams to FlatSM models.

2nd GG: transforms the obtained FlatSM models into a form of CPN called ONM models and relates each other according to the message passing described in Collaboration diagram. The resulted model of this step is a single CPN model for the system under study.

3rd GG: rebuilds the single CPN model in the input language of the INA Petri nets analyzer tool.

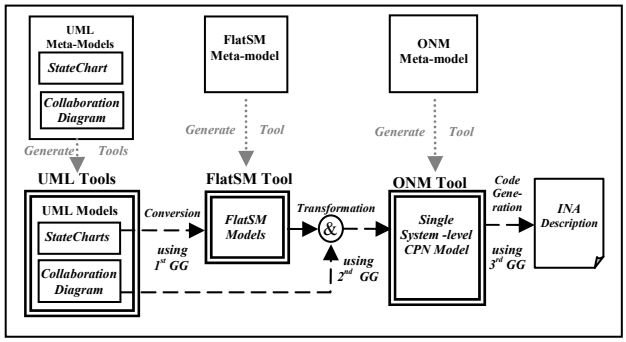

Figure 4. The proposed approach.

A. Meta-Modeling of Used UML Diagrams and Formalisms

To define a modeling language, one has to provide abstract syntax (denoting constructs, their attributes, relationships and constraints) as well as concrete graphical syntax information (the appearance of constructs and relationships in the visual tool). The meta-formalism used in our work is the UML Class Diagram and the constraints are expressed in Python code.

In this paper, we deal with a subset of UML Statechart [6] which consists of states (simple and composite), transitions, events, actions that generate events, and initial and final states. Composite states have nested structure of states which can be sequential or concurrent. The proposed meta-model for UML Statecharts diagram (see Fig.5 (a)) is composed of the following classes:

Statechart: This has a *Name* and represents a Statechart in the diagram.

SC_State: This class describes simple states and has tree attributes, namely *Name*, *EntryAction* and *ExitAction*.

SC_CompositeState: This class represents composite states. It inherits from *SC_State* all its attributes, multiplicities and associations.

SC_Initial: These kinds of entities mark the initial state of a Statechart or the initial state/states when reaching a composite state.

SC_Final: Represents the final state of a Statechart (if any).

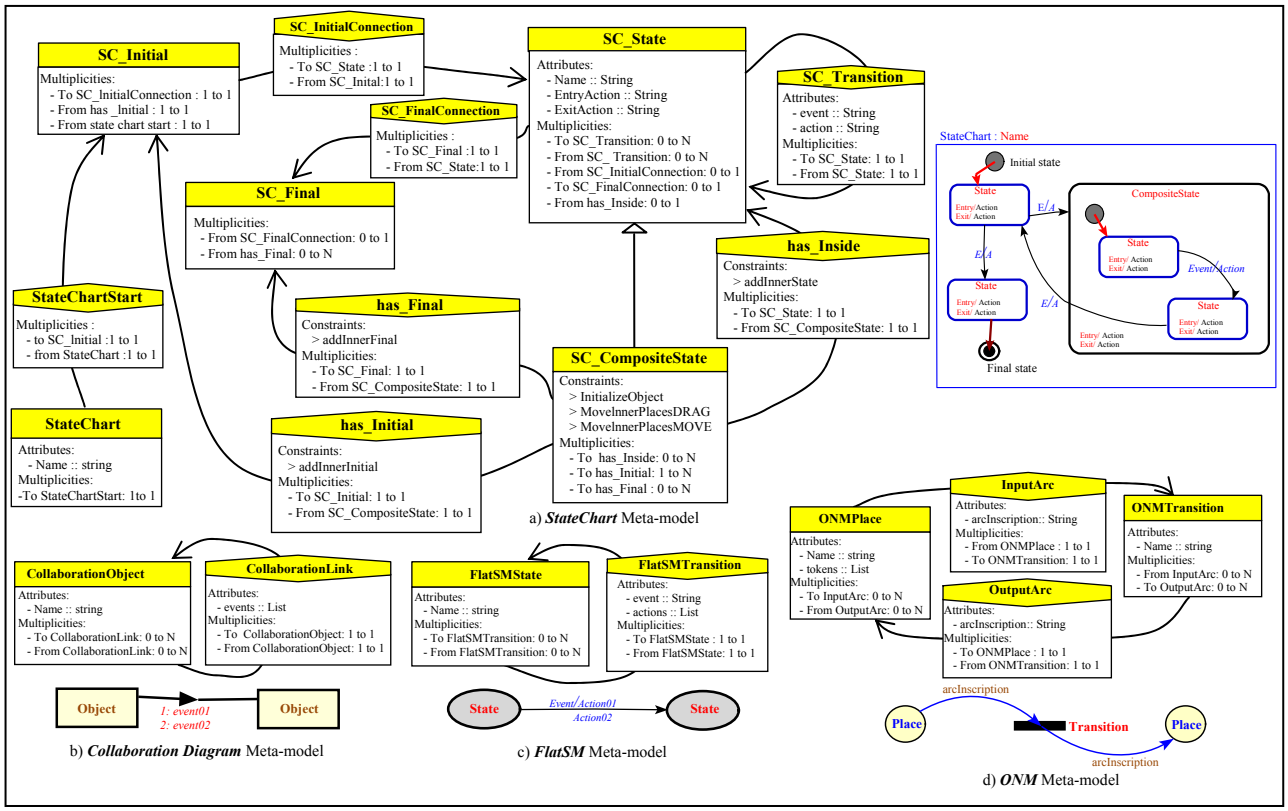

Figure 5. Meta-models of Used UML Diagrams and Formalisms.

The following associations are also included in the meta-model:

StatechartStart: This association allows the connection of a Statechart and its initial state.

SC_InitialConnection: This association allows the connection of an initial state and a state.

SC_FinalConnection: This association allows the connection of a state and a final state.

SC_Transition: This association represents the transition from source state to destination state (which may be the same state). It contains two attributes: *event* and *Action*.

has_Inside: This is an association between a composite state and a state (which may be in it turn composite). It expresses the notion of hierarchy: states are inside composite state.

has_Initial: This association expresses the notion of hierarchy between a composite state and its initial state.

has_Final: This association expresses the notion of hierarchy between a composite state and its final state.

Since UML Collaboration diagram consists of objects that interact by sending each other messages (or events), we propose to meta-model UML Collaboration diagram with one class named *CollaborationObject* for representing objects and one association named *CollaborationLink* for representing communication between two objects through a list of events as shown in Fig.5 (b). A FlatSM is a State Machine without composite states. States in FlatSM are denoted by rounded boxes, while transitions between states are represented with arcs. The Fig.5 (c) presents our meta-model for FlatSM formalism. It consists of one class to represent FlatSM states and one association for representing FlatSM transitions. The last formalism used in our work is ONM. ONM is a form of Object Petri Net which uses places and transitions to make up models. In order to define meta-model for ONM we propose two classes: *ONMPlace* class to describe places and *ONMTransition* to describe transitions. These classes are related with two associations named *InputArc* and *OutputArc* which represent input arcs and output arcs as shown in Fig.5 (d).

To fully define our meta-models, we have also specified the graphical appearance of each entity of the formalisms according to its appropriate graphical notation. Since the associations: *has_Inside*, *has_Initial* and *has_Final* in the Statechart Meta-model are a means to express hierarchy, they are drawn as invisible links.

Given our meta-models, we have used AToM^3 to generate for each formalism a palate of buttons allowing the user to create the entities defined in its meta-model. Since AToM^3 is a visual tool for multi-formalism modeling, we can show in the user interface of AToM^3 all generated tools at the same time (see Fig.11).

B. Graph Transformation Grammars

As we mentioned earlier, graph rewriting systems iteratively apply a list of rules to the host graph. In AToM^3, rules are ordered according to a user-assigned priority, and are checked from higher to lower priority. In the LHS of rules, the attributes of the nodes must be provided with attribute values which will be compared with the nodes attributes of the host graph during the matching process. These attributes can be set to <*ANY*> or have specific values. In order to specify the mapping between LHS and RHS, Nodes in both LHS and RHS are identified by means of labels (numbers). If a node label

appears in the LHS of a rule, but not in the RHS, then the node is deleted when the rule is applied. Conversely, if a node label appears in the RHS but not in the LHS, then the node is created when the rule is applied. Finally, if a node label appears both in the LHS and in the RHS of a rule, the node is not deleted. If a node is created or maintained by a rule, we must specify in the RHS the attributes' values after the rule application. In AToM^3 there are several possibilities. If the node label is already present in the LHS, the attribute value can be copied (<*COPIED*>). We also have the option to give it a specific value or assign it a Python program to calculate the value (<SPECIFIED>), possibly using the value of other attributes.

In this subsection, we use AToM^3 to define the three Graph Grammars for our approach.

1St GG: Converting Statechart diagrams into FlatSM models

We have named this graph grammar *Statechart2FlatSM*. To convert a Statechart into its equivalent FlatSM model using our *Statechart2FlatSM* grammar, we have proposed twenty three rules which will be applied in ascending order. Due to lack of space, only some representative rules are shown in Fig.6 and Fig.7.

The graph grammar has an initial Action which decorates all the composite state and transition elements in the Statechart model with temporary attributes to be used in the matching conditions of rules. In composite state elements, we use two attributes: *Traversed* and *Processed*. The *Traversed* attribute is used for indicating if the composite state has already been traversed by the grammar, whereas the *Processed* attribute indicate whether the composite state has been processed yet. In transition elements, we use also an attribute (*Converted*) which indicates if the transition has already been converted by the grammar. All these temporary attributes are initialized to 0.

The strategy of the conversion in *Statechart2FlatSM* grammar can be summarized by the following main steps:

The first step is to select a Statechart form the set of Statechart diagrams and to convert its initial state into a FlatSM state (rule N°22), More precisely, this rule consists in associating a FlatSM state to the Statechart initial state by a generic link that permit to connect model elements from different language in AToM^3.

The second step is to traverse the selected Statechart through its transitions from the initial state to next states and so forth. The next states can be simple or composite states. For the simple states, these states and the incoming transitions are converted into FlateSM states and FlateSM transitions respectively (rule N°5 and N°6). In the case where the next simple state has been previously processed, it will not be converted more than once. For this raison the rule N°5 has a higher priority than the rule N°6. When the next state is composite state, *Statechart2FlatSM* grammar sets this state as traversed (*Traversed = 1*) without converting the incoming transition as shown in rule N°7. At the end of this step, all Statechart states of the first level of hierarchy are traversed. The simple states are converted, whereas the composite ones are not yet.

In third step, the traversed composite states (if any) will be converted. The conversion strategy is the same for Statechart diagrams which is based on same traversing process recursively (rule N°1). The last step is to relate the equivalent FlatSM segments of the composite states to FlatSM model. For example, rule N°14 converts exit transiotion (not previousely Converted) from composite state to simple state into its equivalent transition in FlatSM model, and so forth for all other levels of hierarchy.

In order to convert a concurrent composite state, the *Statechart2FlatSM* graph creates two spatial FlatSM states (*Fork* and *Joint* states) which mimic the semantic of the concurrence (rule N°20). The *Fork* state denotes parallel activation of all immediate successor states, whereas the *Join* state denotes synchronized activation of an immediate successor state, with respect to join state's source states. The conversion of nested states is performed as described in the first and second steps. Then, all initial nested states will be related to *Fork* state (rule N°18) and all final nested states will be related to *Join* state (rule N°17). For the sequential composite states, the graph grammar locates the initial nested state and creates an equivalent FlatSM state and converted next nested states according the traversing process as described below (in the second step). In Fig.8, we show the conversion of a concurrent composite state into FlatSM form.

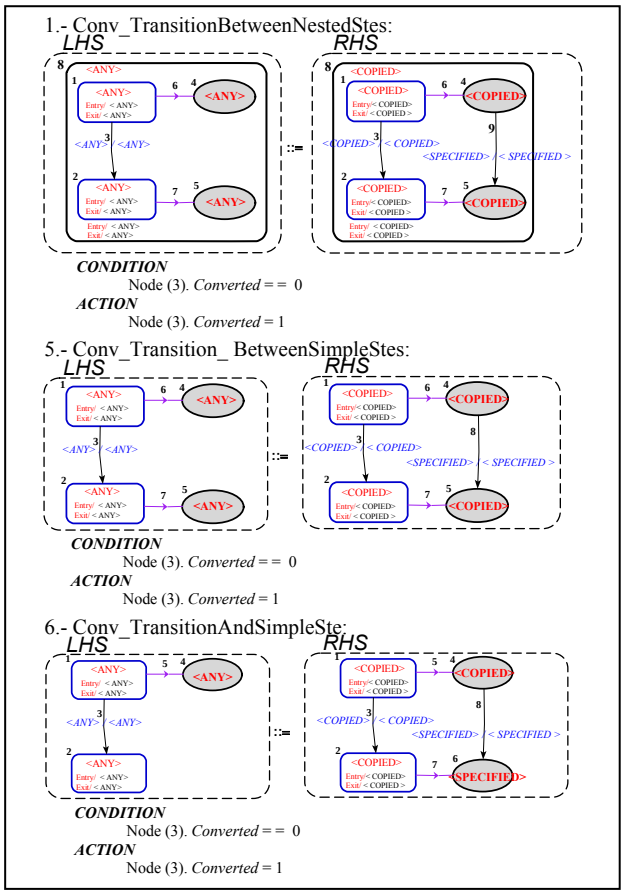

Figure 6. 1st GG Rules: N° 1, N°5 and N°6.

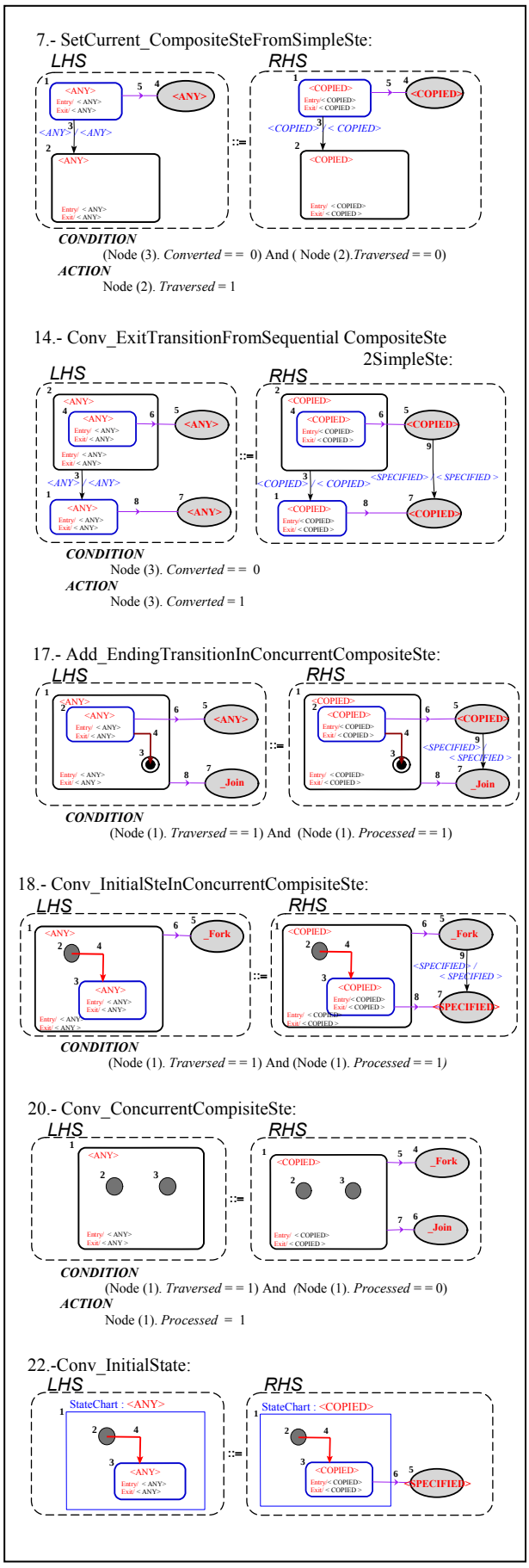

Figure 7. 1st GG Rules: N°7, N°14, N°17, N°18, N°20 and 22.

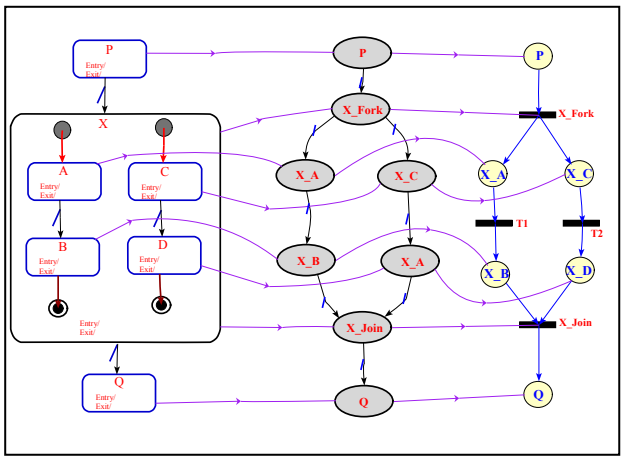

Figure 8. The Conversion of a concurrent composte state (X) to FlatSM form and ONM form.

2nd GG: Constructing Intermediate system-level model

We have named the second graph grammar *FlatSM2ONMs*. When the *FlatSM2ONMs* graph grammar execution finishes, the resulting model is the CPN system-level model. Fig.9 shows the important proposed rules.

The idea behind the transformation can be described in the following steps. In the first one, the graph grammar selects a Statechart and creates both the token routing structure (IP, ED and OP places and OTA, ITA , T1 and T2 transitions) and an equivalent ONM place for its initial state (rule N°14). The second step consists of creating the LM for the selected Statechart based on its equivalent FlatSM model generated in the first graph grammar. The transformation process is based on traversing FlatSM model in the same manner used in the first graph grammar. An ONM place is created for each FlatSM state, and an ONM transition is for each FlatSM Transition (for example, rules N°2). We note that the *Fork* or *Join* states in the FlatSM model will be transformed into ONM transitions (see Fig.8).

At this point, LM should be linked to the token routing structure. For each FlatSM transition that generates events, an output arc will be created from the associated ONM transition to ED place in the token routing structure. If the event is an external event that will be sent to other objects, the created output arc will have an inscription as "ex". The external events of object are specified in the Collaboration diagram (rule N°8). Otherwise, the event will be considered as an internal event (rule N°9). Linking IP place to LM will be processed in same way as for the ED place but with input arc. Then, ITA and OTA transitions are related to ILP place (rule N°11).

Finally, *Statechart2FlatSM* graph grammar deletes all elements of Statechart diagrams, Collaboration diagram and FlatSM formalism in order to have only the system-level model in ONM formalism (for example, rule N°17 for deleting Statechart state).

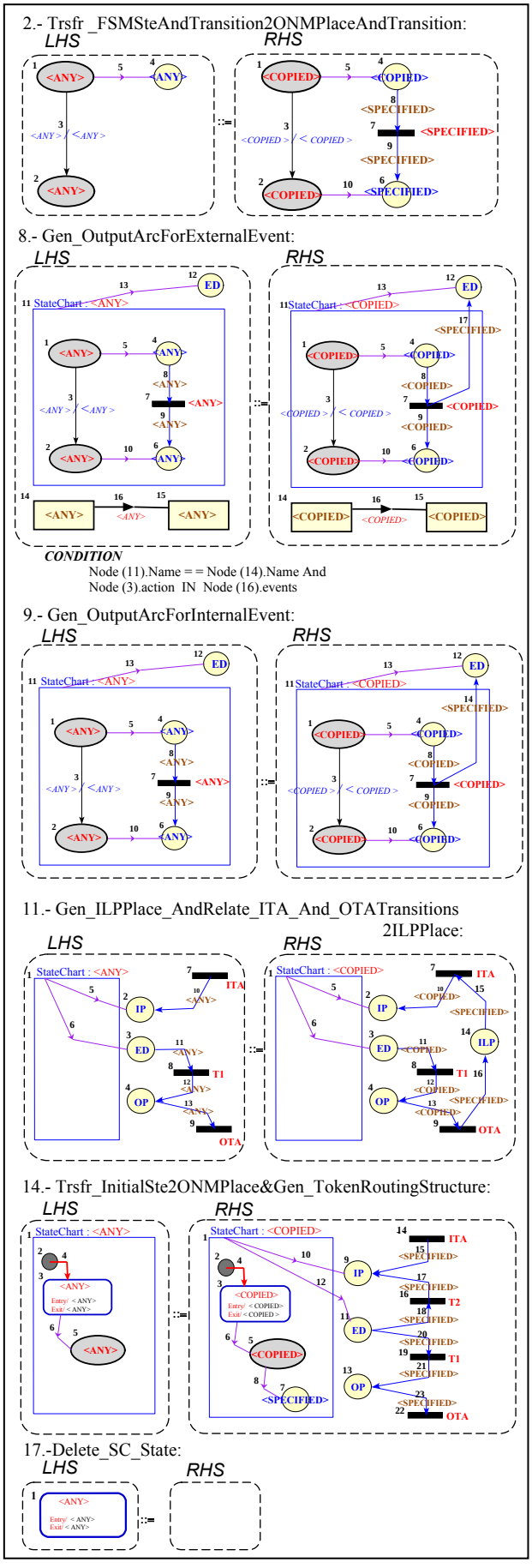

Figure 9. 2nd GG Rules: N°2, N°8, N°9, N°11, N°14 and N°17.

3rd GG: Generating the INA description

We have named the last graph grammar *ONMs2INA*. This grammar generates the INA description file (the .cnt file) of the whole system. The .cnt file consists of the unfolded net structure (which describes the places colors (sub-places), their initial markings, and the relations with the transition colors (sub-transitions)), and the folding information.

The *ONMs2INA* graph grammar has an initial Action which opens the file where the INA code will be generated and decorates all the Place and Transition elements in the model with temporary attributes to be used in the conditions specified in the rules. In place elements, we use two attributes *C* (*Current*) and *V* (*Visited*). The *C* attribute is used to identify the place in the model whose code has to be generated (*C ==1*), whereas the *V* attribute is used to indicate whether code for the place has been generated yet (*V == 1:* for unfolded net structure, *V == 2:* for sub-places information section and *V == 3:* for the aggregation section). In Transition elements, we use three attributes *PRE*, *POST* and *V* (*Visited*). The *PRE* attribute is used to indicate whether this transition is processed as pre-transition for the current sub-place, whereas the *POST* attribute is used to indicate if this transition is processed as post-transition. The *V* attribute is used to indicate whether code for the transition has been generated yet (*V == 1:* for sub-transitions information section and *V == 2:* for the aggregation section). All these temporary attributes are initialized to 0.

The *ONMs2INA* graph grammar is composed of ten rules. We are concerned here by code generation, so none of these rules will change the input model as shown in Fig.10. The main steps of the Automatic code generation can be described as follows:

The first step is to select a place and assign a number for it identification which is unique, and marks it as current. For the sub-places of the selected place, the graph grammar assigns also a number for each of its sub-places (ruleN°6).

The second step consists in generating the unfolded net structure. For each sub-place in the current place, one line will be generated. Each such line starts with the number of the sub-place, followed by one blank and the number of tokens in the initial marking (ruleN°4). If the sub-place has pre-transitions, these are generated as elements of list separated by blanks (ruleN°1). Likewise, if the sub-place has post-transitions then the graph grammar generates a similar list which is preceded by "," (ruleN°2). We note that rule N°3 is used to initializes *POST* and *PRE* attributes of all transitions for the processing of the next sub-place. Whereas, rule N°5 is used to locate the current place whose processing has been terminated (for all its sub-places), and mark it as Visited (*V == 1*). This process will be repeated for all others not visited places in the CPN model (ruleN°6 for selecting anther not visited place).

The last step is to generate the folding information, i.e., the information which colored places and transitions exist, and what their colors are. The rules N°7 and N°8

generate the sub-places and sub-transitions information sections respectively. The assigned number followed by the color which represents the event-name are generated for all sup-places and for all sub-transitions. Whereas, the rules N°9 and N°10 generate the places and transitions aggregations information section respectively. Each place will be defined by generating its number, name and color set which is a set of its sub-places numbers. For every transition, the number, name and color set will be also generated.

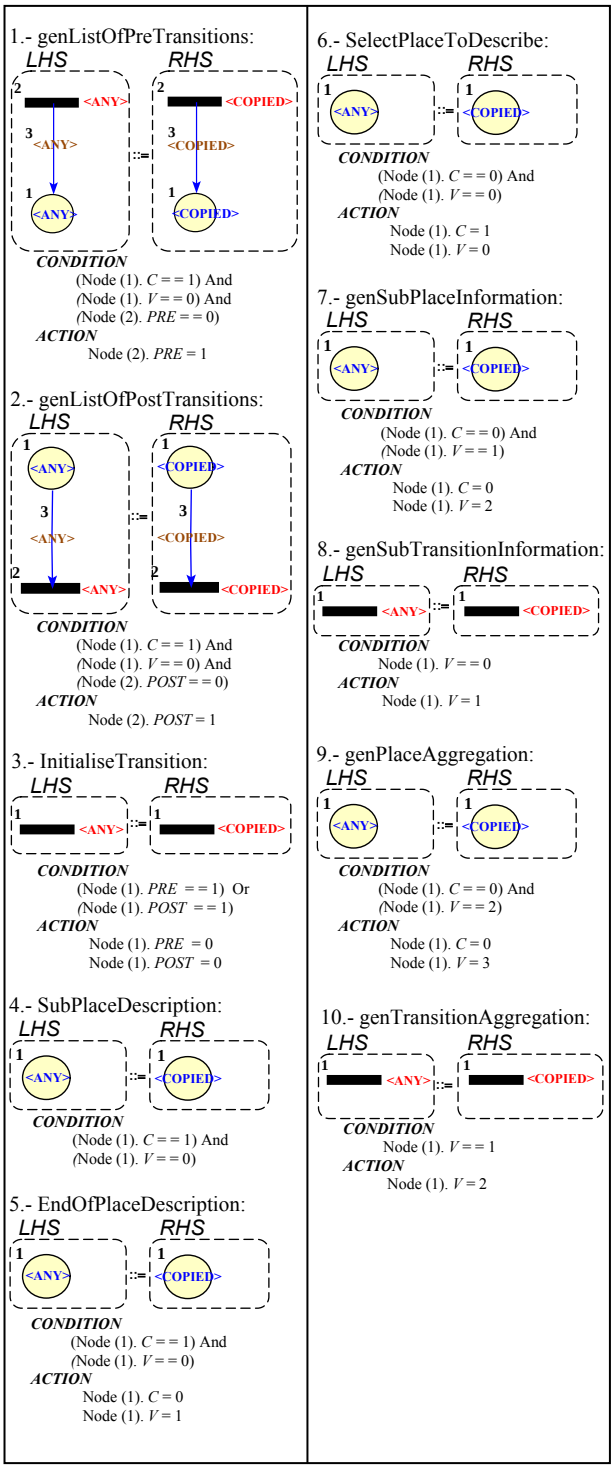

Figure 10. 3rd GG: Generating INA specification.

VI. Case Study: ATM Machine

We consider an example of an ATM machine, dispensing cash to a user [6]. The Statechart diagrams and the Collaboration diagram created in our tools are shown in Fig.11 (in the top of the canvas tool). The description of the problem is as follows: An ATM machine has three basic states: Idle (waiting for customer interaction), Active (handling a customer transaction) and Maintenance (perhaps having a cash store replenished). While active, the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction and then print a receipt. After printing, the ATM returns to the idle state. While in the active state, the user might any time cancel the transaction, returning to the ATM to the idle state. The Statechert diagram of ATM user shows the different user's actions. All named events in the Statechart diagrams, which do not show up on the collaboration diagram, form the internal events of that respective object.

In order to analyze this behavioral specification of the ATM Machine, we have to transform this specification into its equivalent system-level CPN model. To realize this transformation in our tools, we have to execute the *Statechart2FlatSM* graph grammar to obtain flattened FlatSM models from composite Statechart diagrams as shown in Fig.11 (in the bottom of the canvas tool). Then we have to execute also the *FlatSM2ONMs* graph grammar to synthesize the system-level CPN model from the obtained FlatSM models and Collaboration diagram. The resulted CPN model of the automatic transformation is shown in Fig.12.

In order to perform the analysis of the resulted CPN model using the INA analyzer we have to generate its equivalent INA description. To generate INA description in our tool, we have to execute the *ONMs2INA* graph grammar defined in the previous section. A part of the automatic generated file *SYSTEM-LEVEL_MODEL.cnt* which contains the INA description of ATM machine is shown in Fig.13.

To analyze the properties of the behavioral specification of the ATM Machine, we have invoked the INA tool with the generated INA specification file as input. Then, the INA tool provides the properties of the Petri Net as shown in Fig.14. We can see from INA screen that the net is not bounded, not live, not safe and the deadlock-trap property is not valid.

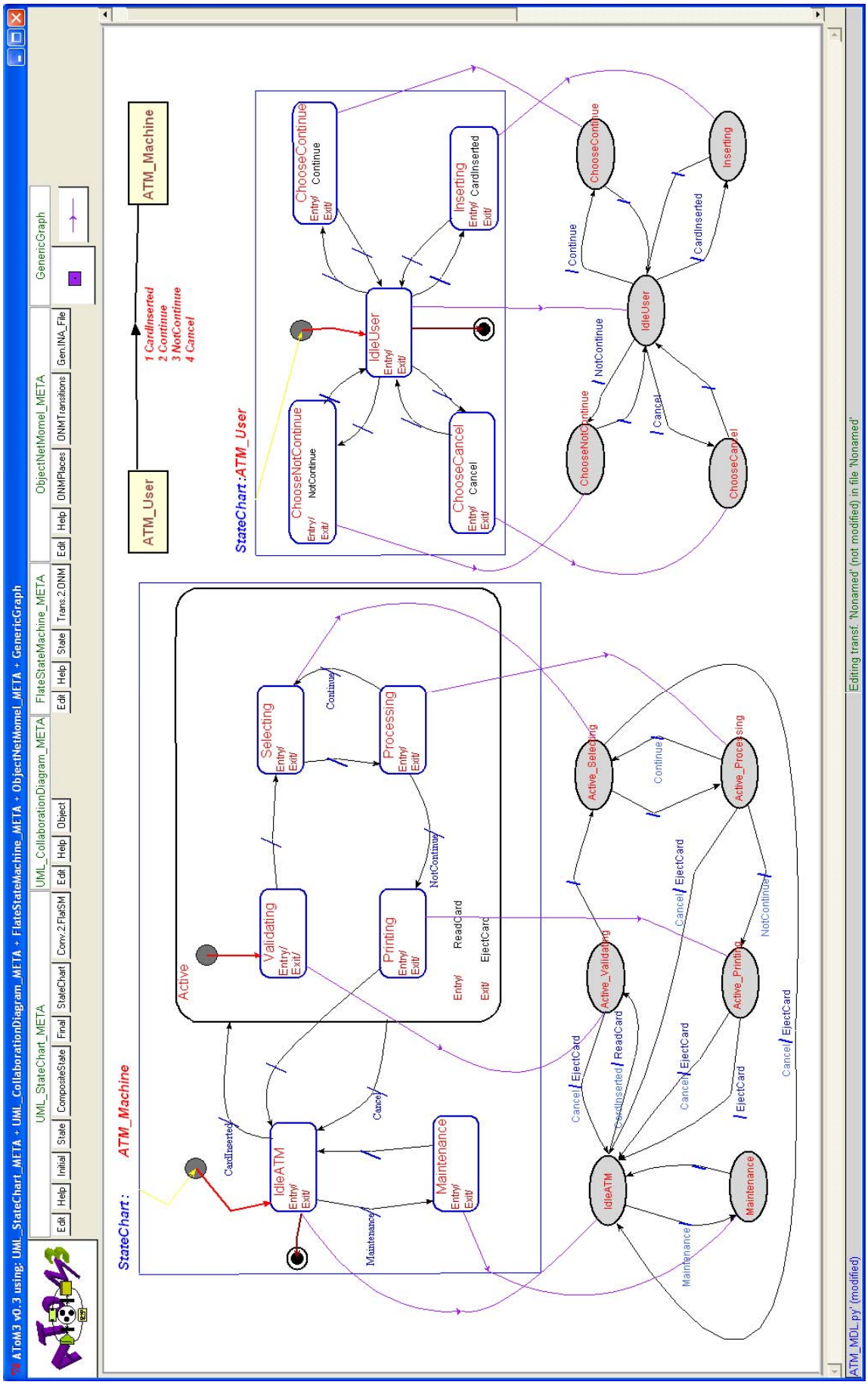

Figure 11. Behavioral specification of ATM Machine.

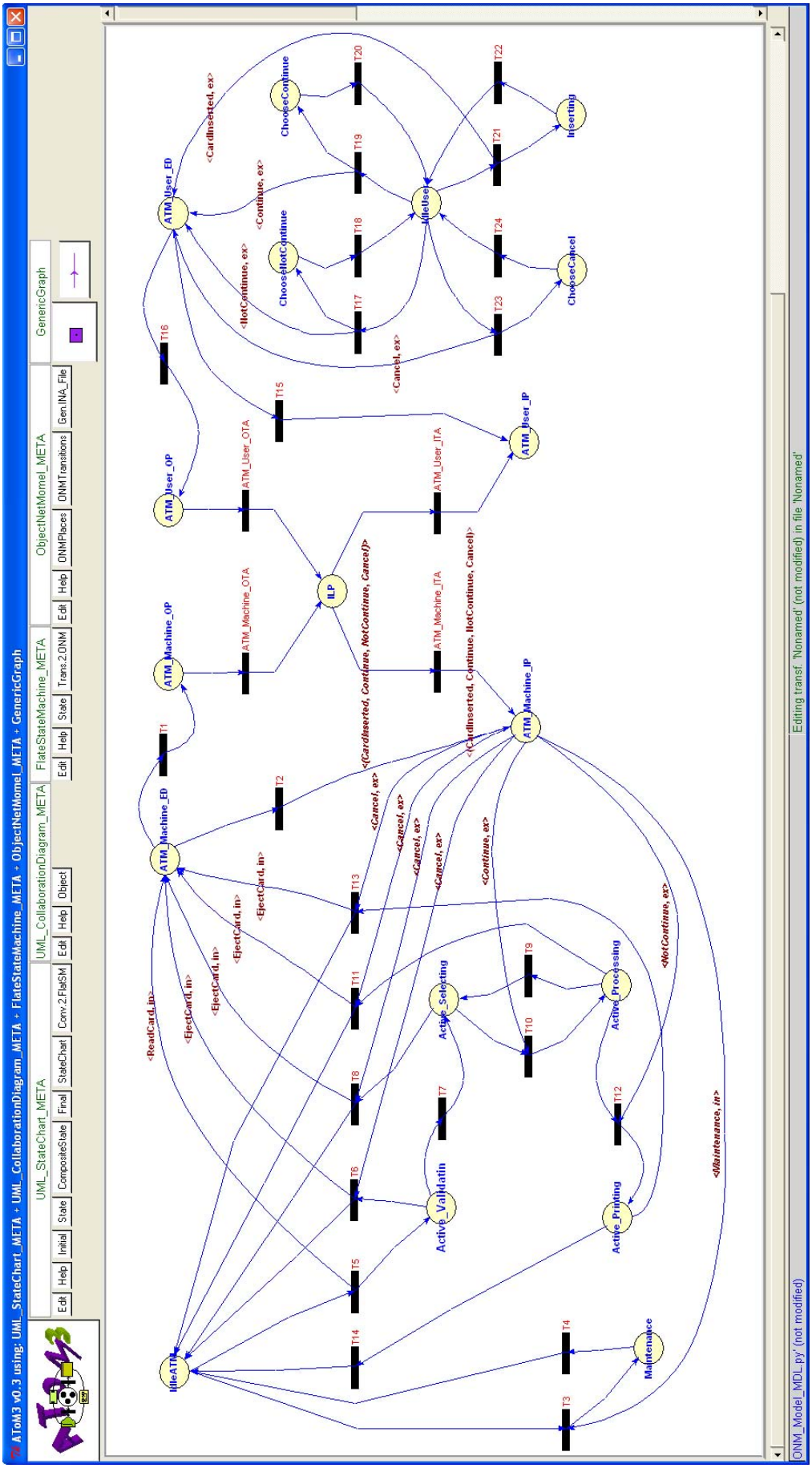

Figure 12. System-level CPN model of ATM Machine.

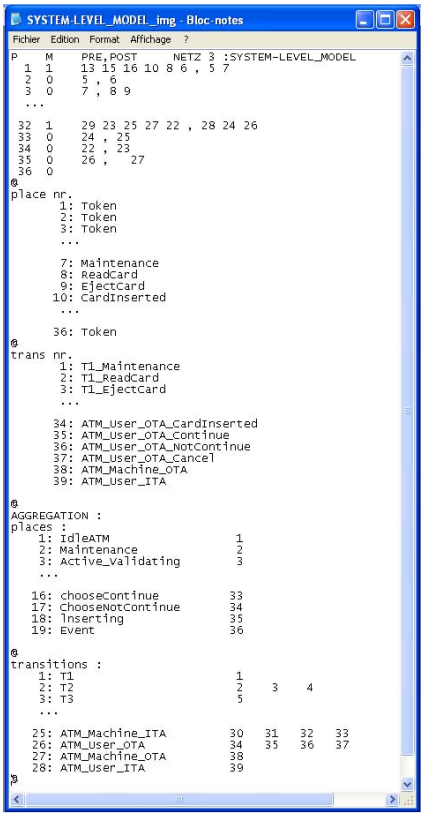

Figure 13. Generated INA specification.

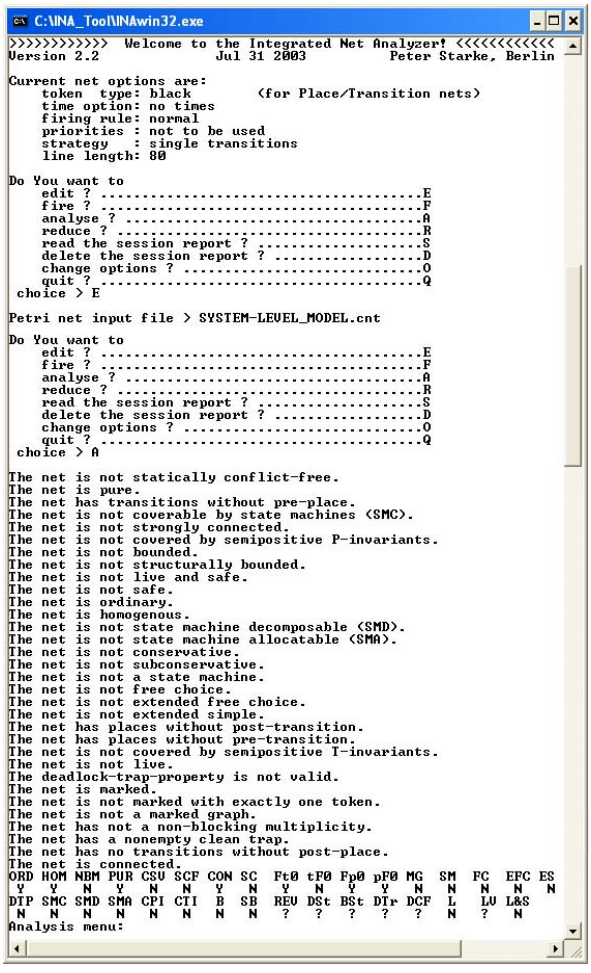

Figure 14. Analysis of the obtained System-level CPN model.

VII. CONCLUSION

In this paper we have proposed a Graph transformation approach for transforming UML Statechart and Collaboration diagrams to Colored Petri nets models. More precisely, we have proposed an automated approach and tool environment that formally transform dynamic behaviors of systems expressed using UML models into their equivalent Colored Petri Nets (CPN) models for analysis purposes. This transformation aimed to bridge the gap between informal notation (UML diagrams) and more formal notation (colored Petri nets models). It produces highly-structured, graphical, and rigorously-analyzable models that facilitates early detection of errors like deadlock, live-lock, … .To make the analysis easier, we have used the obtained CPN models to generate automatically their equivalent description in the input language of the INA Petri net analyzer. Our approach is based on graph transformation and the meta-modeling tool ATOM^3 was used. We have illustrated our approach through an example.

In future work we plan to transform other UML diagrams to colored Petri nets and use the well known reduction technique on the obtained models before performing the analysis in order to optimize the models. We plan also to back-annotate the analysis results into the UML diagrams to reach the complete automation of the transformation.

REFERENCES

[1] AGG Home page, http://tfs.cs.tu-berlin.de/agg/
[2] AToM^3 Home page, http://atom3.cs.mcgill.ca/
[3] R. Bardohl, H. Ehrig, J. De Lara and G. Taentzer, ″Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation″, Proc. Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS Springer, Heidelberg, Vol. 2984, pp. 214–228, 2004.
[4] S. Bernardi, S. Donatelli and J. Merseguer. "From UML sequence diagrams and statecharts to analysable petri net models". Proc. 3rd international workshop on Software and performance, pp. 35-45, Rome, Italy, 2002.
[5] D. Blonstein, H. Fahmy and A. Grbavec. ″Issues in the Practical use of Graph Rewriting″. LNCS, Vol. 1073, Springer pp.38-55, 1996.
[6] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.
[7] J. De Lara and H. Vangheluwe, ″AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling″, Proc. ETAPS/FASE'02, LNCS, Vol. 2306, pp.174-188, 2002.
[8] J. De Lara and H. Vangheluwe, ″Computer Aided Multi-Paradigm Modelling to Process Petri-nets and Statecharts″, Proc. International Conference on Graph Transformations (ICGT), LNCS, Vol. 2505, pp. 239-253Springer-Verlag, Barcelona, Spain, 2002.
[9] J. De Lara and H. Vangheluwe, ″Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3″, Software and Systems Modelling, Special Section on Graph Transformations and Visual Modeling Techniques, Vol. 3, pp. 194–209, 2004.
[10] Y. Deng, S.K. Chang, J. De Figueired and A. Psrkusich. ″Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex

Information Systems″. Proc. The 14th International Conference on Application and Theory of Petri Nets, LNCS, Vol. 691, pp. 206-223, Chicago, 1993.
[11] Z. Dong and X. He, "Integrating UML Statechart and Collaboration Diagrams Using Hierarchical Predicate Transition Nets", Lecture Notes in Informatics, Workshop of the pUML-Group held together with the «UML»2001 on Practical UML-Based Rigorous Development Methods, Vol. 7, pp. 99-112, 2001.
[12] R. El Mansouri, E. Kerkouche and A. Chaoui, ″A Graphical Environment for Petri Nets INA Tool Based on Meta-Modelling and Graph Grammars″, Proc. World Academy of Science, Engineering and Technology, ISSN 2070-3740, Vol. 34, pp.471-475, 2008.
[13] EMF Home page, http://www.eclipse.org/emf/
[14] FUJABA Home page, http://www.fujaba.de/
[15] GEF Home page, http://www.eclipse.org/gef/
[16] H.J. Genrich and K. Lautenbach. ″System Modelling with High-Level Petri Nets ″. Theoretical Computer Science, Vol. 13, pp. 109-136, 1981.
[17] GME Home page, http://www.isis.vanderbilt.edu/gme/
[18] GMF Home page, http://www.eclipse.org/gmf/
[19] GReAT Home page, http://www.escherinstitute.org/Plone/tools/
[20] D. Heral, ″Statechart: A Visual Formalism for Complex Systems″, Science of Computer Programming. Vol.8, pp.231-274. 1987.
[21] INA Home page, http://www2.informatik.hu-berlin.de/~starke/ina.html
[22] K. Jensen, Coloured Petri Nets, Vol. 1: Basic Concepts, Springer-Verlag, 1992.
[23] S. Kelly, K. Lyytinen and M. Rossi, ″MetaEdit+: A fully configurable Multi-User and Multi-Tool CASE and CAME Environment″, Proc. Constantopoulos, P., Vassiliou, Y., Mylopoulos,J. (eds.) CAiSE 1996. LNCS Springer, Heidelberg, Vol. 1080, 1996. MetaEdit+ Homepage, http://www.MetaCase.com
[24] E. Kerkouche and A. Chaoui, ″A Formal Framework and a Tool for the Specification and Analysis of G-Nets Models Based on Graph Transformation″, Proc. International Conference on Distributed Computing and Networking - ICDCN'09-, LNCS Springer-Verlag Berlin Heidelberg,, Vol. 5408, pp. 206–211, India, 3-6 January, 2009.
[25] E. Kerkouche, A. Chaoui, E. Bourennane, and O. Labbani: "Modelling and verification of Dynamic behaviour in UML models, a graph transformation based approach", proceedings of SEDE'2009, Las Vegas, Nevada, USA, 22-24 June 2009.
[26] KerMeta Home page, http://www.kermeta.org/
[27] C. Lakos, "Object Petri Nets - Definition and Relationship to Colored Petri Nets", Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.
[28] T. Murata, ″Petri Nets: Properties, Analysis and Applications″, Proc. IEEE, Vol. 77, pp. 541-580, No.4, 1989.
[29] PROD Home page, version 3.4.01, http://www.tcs.hut.fi/Software/prod/
[30] PROGRES Home page, http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html
[31] Python Home page, http://www.python.org
[32] G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1. World Scientific, Singapore, 1999.
[33] J.A. Saldhana, S. M. Shatz and Z. Hu, ″Formalisation of Object Behavior and Interaction From UML Models″, International Journal of Software Engineering and Knowledge Engineering. Vol. 11, pp. 643-673, 2001.
[34] S. Shlaer and S. J. Mellor, Object Life Cycles Modeling the World in States, Yourdon Press, Prentice Hall, 1992.
[35] TIGER Home page, http://tfs.cs.tu-berlin.de/tigerprj/

Elhillali Kerkouche is Assistant Professor in the department of Computer science, University of Oum El bouaghi, Algeria. His research field is formal methods and Distributed Systems.

Allaoua Chaoui is with the department of computer science, Faculty of Engineering, University Mentouri Constantine, Algeria. He received his Master degree in Computer science in 1992 (in cooperation with the University of Glasgow, Scotland) and his PhD degree in 1998 from the University of Constantine (in cooperation with the CEDRIC Laboratory of CNAM in Paris, France). He has served as associate professor in Philadelphia University in Jordan for five years and University Mentoury Constantine for many years. During his career he has designed and taught courses in Software Engineering and Formal Methods. Dr Allaoua Chaoui has published many articles in International Journals and Conferences. He supervises many Master and PhD students. His research interests include Mobile Computing, formal specification and verification of distributed systems, and graph transformation systems.

El Bay Bourennane is Professor of Electronics at the laboratory LE2I, (Laboratory of Electronics, Computer Science and Image). University of Bourgogne, Dijon, France.

Ouassila Labbani is an Associate Professor at University of Bourgogne, Dijon, France. Her research field is UML and formal methods.