A Transformation Approach for Modeling and Analysis of Complex UML Statecharts: A Case Study

Zhaoxia Hu
Department of Computer Science
University of Illinois at Chicago
Chicago, IL, U.S.A.

Sol M. Shatz
Department of Computer Science
University of Illinois at Chicago
Chicago, IL, U.S.A.

Abstract - We have previously proposed a Petri-net based transformation approach to support formal modeling and analysis of complex UML statecharts. The generation of the target net model is based on an intermediate model. In this paper we define steps for generating the target model from the intermediate model, and focus on a case study to evaluate the overall approach. The case study is based on modeling and simulation of an Early Warning System and a set of four experiments.

Keywords: UML statecharts Formal methods transformation Petri nets model analysis

1.0 Introduction

The Object Management Group (OMG) adopted a new paradigm for software development called Model Driven Architecture (MDA) to recognize the fact that models are important artifacts of software development and they serve as a basis for systems as they evolve from requirements through implementation. In MDA, models are defined in the Unified Modeling Language (UML) [2]. The lack of formal dynamic semantics for UML limits its capability for analyzing defined specifications. Many research efforts have been carried out in the area of formalizing UML by mapping the UML notation to a formal notation to give the UML notation a precise semantics and achieve UML verification [14]. We have previous proposed a UML-CPN transformation framework [6, 13] to introduce dynamic model analysis into UML modeling by mapping UML models to Petri net models, in particular colored Petri nets (CPNs) [8]. Our approach is concerned with transformation of one core component of UML – statechart diagrams [10].

In this paper, we discuss the key issues for transformation and analysis of complex statecharts. By complex statecharts, we mean statecharts that contain composite states. Composite states complicate the transformation of UML statecharts to other modeling languages because the semantics associated with composite states are often not explicitly observed in statecharts. To decompose the transformation problem, we introduced an intermediate model [7] that solves hierarchical, history and composition related problems in UML statecharts before running a more straightforward translation to a state/event based formalism, here CPN. In this paper, we present steps for converting our intermediate model into the target notation supported by Design/CPN, and we focus on a case study to evaluate the overall approach. This case study includes a set of experiments to reason about the model's behavior. There are related efforts in the area of modeling to support validation and analysis of UML statecharts [1, 3, 4, 9, 11, and 12]. Since the emphasis of this paper is not on the approach but on the case study, we do not present details on such related work. Further details can be found in [7].

2.0 An intermediate model for composite states

To make this paper self-contained, we begin by summarizing some of the key translation concepts that have previously been defined [7]. Our intermediate model is introduced to make explicit the "implied" semantics associated with composite states. C*ontrol-states* are introduced so that the control flow of the state machines is explicitly represented in the intermediate model.

Entry transition. The basic form of the intermediate model for entry transitions of a composite state is a fork transition as shown in Fig. 1. A fork transition has one source state and multiple target states. We call this special fork transition an *init transition*. An *init* transition uses a trigger to model the triggering-condition for an entry transition. The trigger is the triggering event of the UML entry transition. The source state of the *init* transition is the same as that of the UML entry transition. The target states of the fork transition include all the states that are to be activated by this entry transition, including the composite state itself and some appropriate nested states. To determine the appropriate target states, semantic rules were defined based on whether the transition is a *boundary entry transition* or a *cross-boundary entry transition* [7]. A boundary entry transition targets a composite state while a cross-boundary entry transition targets nested states of a composite state.

Exit transitions. The basic form of the intermediate model for an exit transition is a set of transitions as shown in Fig. 2. The set of transitions consists of an *initial* transition for recognizing that the exit transition is enabled, a *deactivation* module for deactivating the source states, and an *activation* transition for activating the target state. The structure of the deactivation module can be found in previous work [7]. Control states are depicted as ovals in our intermediate models, as shown in Fig. 2.

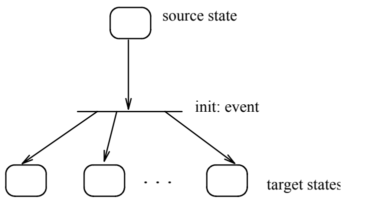

Fig. 1 The intermediate model for entry transitions

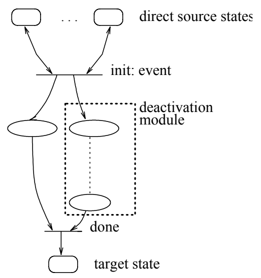

Fig. 2 The intermediate model for exit transitions

History states. To model a history state, which is contained in a region, the idea of a *shadow state* is introduced for each nested state of the region. If some nested state is active when an exit transition fires, the associated shadow state is activated to remember the history. Then when the region is entered because some entry transition targeting the history state fires, the most recently active substate of the region is entered based on the recorded history. Consequently, the translation of exit transitions needs to be modified in two ways: 1) The deactivation module needs to be modified to remember the history via shadow states; 2) A "clearing-history" module needs to be added to the intermediate model for an exit transition for the purpose of reset the shadow states before recording the history. Furthermore, the intermediate model for an entry transition that targets a history state needs to be changed to a 2-level structure. The structure of the top level is similar to an *init* transition for an ordinary entry transition. Transitions at the lower level are defined to model that the nested state associated with the currently active shadow state is entered.

Event dispatching mechanism. The abstract event dispatching model is shown in Fig. 3. This model is designed to address the following issues: 1) how an event-token in the event queue is selected and dispatched; 2) how the dispatched event-token, also called the *current* event-token, is made available to net transitions that may or may not be associated with composite states; and 3) transition priority, i.e., transitions originating from nested states of a composite state, call them *cross-boundary exit transitions*, have higher priority than those originating from the boundary of the composite state, call them *boundary exit transitions*. In this model, the event queue is modeled by a colored token held within the *IP* place, rather than by the *IP* place itself. Place *DE* holds the currently dispatched event-token selected from the event queue. *DE* is connected to ordinary simple transitions and entry transitions contained in the composite state. Place *Step* is introduced to control the dispatching, i.e., to initiate a run-to-completion step [10]. Other places are introduced to make the event-token available to the nested transitions and exit transitions associated with the composite state.

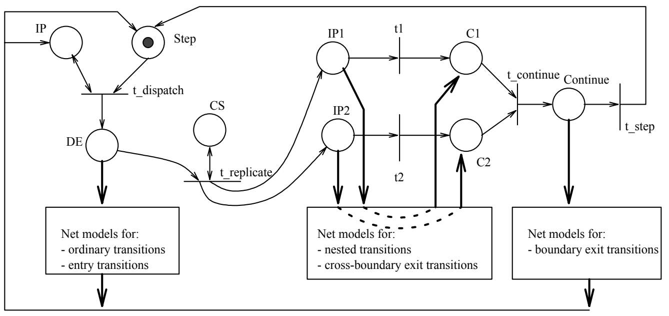

Fig. 3. The abstract event dispatching model

3.0 Target model generation and simulation

To show that the transformation approach can be realized by some existing tool, we choose the Design/CPN [DCPN] tool as an underlying engine to support analysis and simulation of UML diagrams. The high-level steps for constructing the target model, supported by Design/CPN, from the intermediate model are as follows:

1. Since our target model is a colored Petri net, a declaration node is defined for declaring color sets for places and defining variables and functions for arc inscriptions. Details on this aspect of the model are beyond the scope of concern for this paper.
2. For each composite state, the following steps are performed to construct the net model for the composite state. Note that the statechart itself is considered as a (sequential) composite state.
   - Create Petri net places and transitions corresponding to simple states and transitions within the composite state.
   - Define an event dispatching model based on the abstract event dispatching model.
   - Define target models based on intermediate models associated with entry and exit transitions.

The generated model can be imported into Design/CPN for simulation. Design/CPN provides a generic facility to save simulation reports, but the automatically generated reports are not straightforward in terms of providing an end-user with domain-specific information. So, we extend the idea by generating self-defined traces by using *code segments* as supported by Design/CPN. A *code segment* is a sequential piece of code that is defined for a Petri net transition and executed each time the transition occurs. We define code segments for recording the following information to a simulation trace: the object, source states, target states, the triggering event.

4.0 Case study: An Early Warning System

In this section, we show how to translate a complex statechart into the target notation by using the intermediate model. We then run various experiments on the generated model to reason about the behavior of the original UML model. The case study selected incorporates multiple composite states, a history state, and various forms of entry and exit transitions.

4.1 A source statechart

Consider the statechart in Fig. 4, adapted from the early warning system (EWS) model [5]. Although this statechart is a classical statechart, we interpret it as a UML statechart. In order to illustrate the presented approach for translation of history states, we added a history state for region *MONITORING* and a cross-boundary entry transition pointing to this history state. The default history state is state *Waiting_for_command*. The basic behavior of this model can be described as follows. The model consists of a composite state, *EWS_CONTROL*, which is decomposed into two substates. One of the substates, *ON*, is also a composite state and is decomposed into two concurrent regions, *MONITORING* and *PROCESSING*. So, when state *ON* is active, the system is in two states simultaneously, each from a different region. For example, when state *ON* becomes active through the firing of transition *T1*, the system is in the two substates, *Waiting_for_command* and *Disconnected*. Region *MONITORING* contains a sequential composite state, *ACTIVE*, and region *PROCESSING* also contains a sequential composite state, *CONNECTED*. The model also depicts events that cause transitions, such as *reset*, which causes the system to leave both *Comparing* and *Generating_alarm* and enter state *Waiting_for_command.*

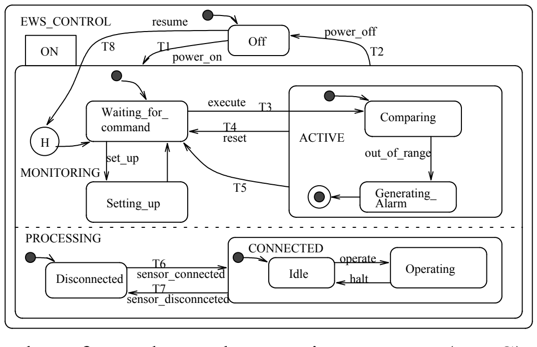

Fig. 4. A statechart from the early warning system (EWS) model

4.2 The intermediate model

For this discussion, we will consider only the outermost composite state, *EWS_CONTROL*. The ideas would equally apply to the other (nested) composite states. We (manually) apply the rules defined in [7] to derive intermediate model. There are two entry transitions, *T1* and *T8*. The intermediate models, denoted as *T1'* and *T8'*, respectively, are shown in Fig. 5. Since transition *T8* targets a history state, model *T8'* has a 2-level structure as mentioned in Section 2.

State *EWS_CONTROL* contains one exit transition, *T2*. The structure obtained from translation of *T2*, labeled with *T2'*, is shown in Fig. 6. An initial transition is used to recognize the enabling condition of transition *T2*, i.e., state *ON* is active, and event *power-off* has occurred. Module 1 is the clearing-history module, and module 2 is the deactivation module. The details of these two modules are not shown in this paper due to lack of space. Finally, an activation transition activates the target state, *Off*.

The abstract event dispatching model for each composite state in the statechart is similar as that shown in Fig. 3.

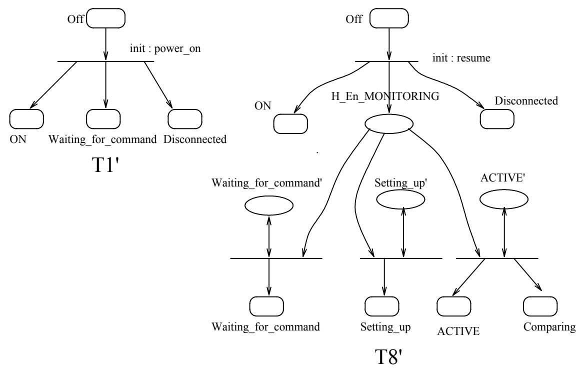

Fig. 5 Intermediate models for entry transitions of state *EWS_CONTROL*

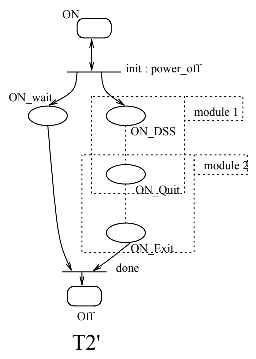

Fig. 6 Intermediate model for exit transition of state *EWS_CONTROL*

4.3 The target model

In this section, we describe how to follow the steps defined in Section 3 to construct the target model for the EWS system based on the original statechart and the intermediate models presented in Section 4.2. We also show how to define specific constructs for the target model to generate simulation traces. As in Section 4.2, we restrict our attention to the composite state *EWS_CONTROL*.

Step 1: The declaration node is defined based on a template declaration node parameterized with the event names specified in the statechart and some variables. In particular, the declaration node defines three color sets, *Active*, *TokenValue*, and *EventList*, among which *TokenValue* is specified with a list of event names defined in the case study statechart. We do not show the resulting declaration node due to lack of space.

Step 2: Create the target net model for the composite state *EWS_CONTROL*. There are two entry transitions, *T1* and *T8*, and one exit transition *T2*. An event dispatching model is defined based on the abstract event dispatching model in Fig. 3. This event dispatching model is shown in the top portion of Fig. 7. The bottom portion of Fig. 7 shows three net structures representing the target models for transitions *T1*, *T8*, and *T2*. These target models are derived from the intermediate models shown in Figures 5 and 6 by adding syntax supported by Design/CPN. Furthermore, the target models include details on obtaining event-tokens dispatched by the event dispatching model and recording the completion of a UML-level transition. For example, in Fig. 7, the net transition for transition *T1* is labeled with *T1:init*. This structure is derived from *T1'* in Fig. 5. Besides the basic structure of *T1'*, this net transition also involves two additional places, 1) place *DE* as an input place to provide an event-token to trigger the transition, and 2) place *Step* as an output place to record the completion of the UML-level transition *T1*. The net transitions for *T2* and *T8* are derived similarly. In UML, if a dispatched event does not trigger any transition, then the event is "discarded." Accordingly, in the model of Fig. 7, transitions *discard* and *new_step* are responsible for discarding unused event-tokens. For clarity, most places are replicated in Fig. 7[1]. There are three other composite

[1] A replicated place is labeled by a rectangle with letter "F" inside.

states in the *EWS* statechart: *ON*, *ACTIVE*, and *CONNECTED*. The target net models for these three states can be constructed similarly, but are not shown here due to space limitations.

Step 3: Define code segments for net transitions to generate simulation traces. The target model contains two primary classes of net transitions – those that directly correspond to *UML transitions* and those that support the modeling of UML transitions. It is the transitions in the first category that are critical for generating simulation traces. We call these transitions *critical transitions*. For example, transition *T1:init* in Fig. 7 is a critical transition since transition *T1:init* is directly derived from transition *T1* of the statechart. On the other hand, transitions *discard* and *new_step* belong to the second category of net transitions; they are support transitions. In this case, *discard* and *new_step* are responsible for discarding unused event-tokens. For each critical transition, a code segment is defined for recording the firing of the transition. A code segment is labelled by a square with letter "C" inside.

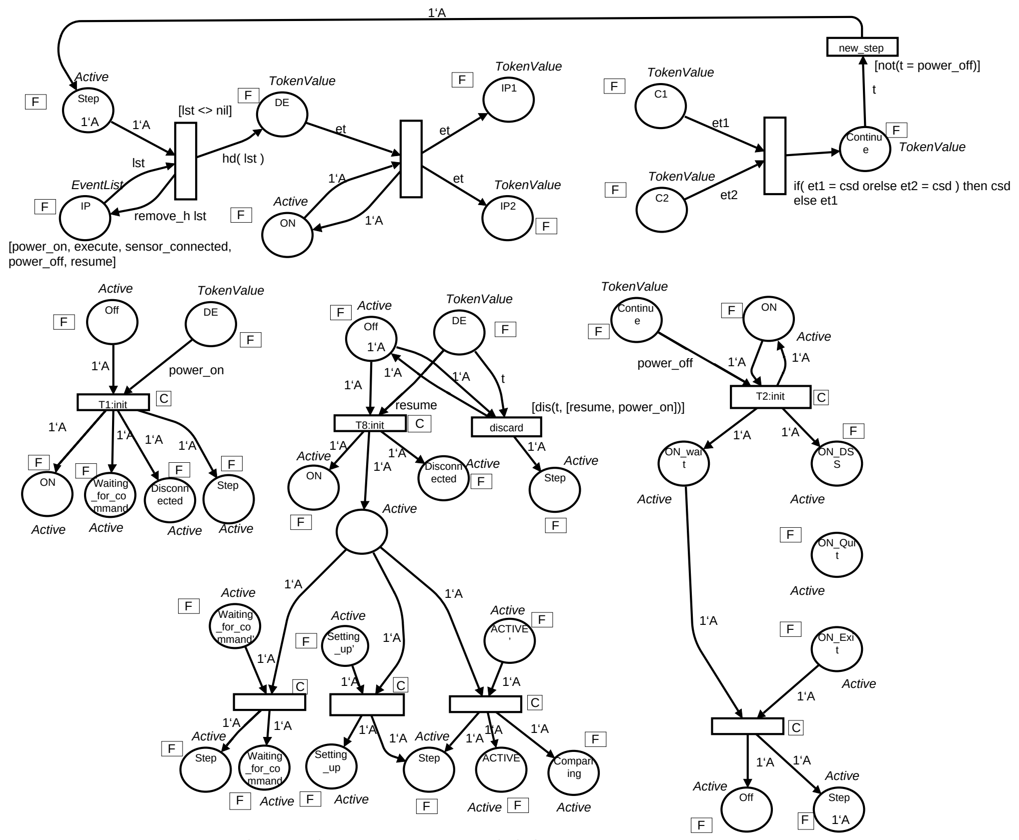

Fig. 7 The target net model for state *EWS_CONTROL*

4.4 Model simulation and analysis

Now that we obtained the target net model, we can use the automated tool Design/CPN to perform experiments on the generated model. In this section, we describe four experiments for checking the generated model's behavior.

Experiment 1. The purpose of this experiment is to test if the history state is modeled correctly. Consider the following scenario. The system is in its default state *Off*. A sequence of events, *power_on*, *execute*, *sensor_connected*, *power_off*, and *resume*, occurs in that order. According to UML semantics, transitions *T1*, *T3*, *T6*, *T2* and *T8* will be enabled and fire in that order. The state change associated with each firing is as follows.

T1: Since *T1* is a boundary entry transition, *T1* deactivates state *Off*, activates states *ON* and its default states (*Waiting_for_command*, and *Disconnected*).

T3: Since *T3* is a cross-boundary entry transition targeting state *Comparing*, *T3* deactivates state *Waiting_for_command*, activates state *ACTIVE* and the direct target state (*Comparing*).

T6: Since *T6* is a boundary entry transition, *T6* deactivates state *Disconnected* and activates states *CONNECTED* and the default state (*Idle*) associated with *CONNECTED*.

T2: Since *T2* is a boundary exit transition, *T2* deactivates state *ON* and the currently active nested states (*ACTIVE*, *Comparing*, *CONNECTED*, and *Idle*) associated with state *ON*, and activates state *Off*.

T8: Since *T8* is a cross-boundary entry transition targeting a history state, *T8* deactivates state *Off*, activates states *ON*, one default state *Disconnected*. Furthermore, since state *ACTIVE* was currently active when composite state *ON* was deactivated by the firing of *T2*, *T8* activates state *ACTIVE* and the default state *Comparing* associated with state *ACTIVE* since *T8* targets the history state.

To check if the model behaves as expected, we did the following experiment. We set the initial marking of the *IP* place as a list of event-tokens, which represented that the event queue held a sequence of events: *power_on*, *execute*, *sensor_connected*, *power_off*, and *resume*. We ran the net simulation, and recorded the following simulation trace:

```
1.  Off -> ON ( Waiting_for_command, Disconnected), trigger: power_off;          // entry transitior T1
2.  Waiting_for_command -> Active( Comparing);  trigger: execute;               // entry transition T3
3   Disconnected -> CONNECTED (Idle); trigger: sensor_connected;                // entry transition T6
4.  leaving ON;   trigger: power_off;  entering Off;                            // exit transition T2
5.  Off -> ON (H, Disconnected);   trigger: resume;  H -> ACTIVE( Comparing);   // entry transition T8
```

From the trace, we can see that the transitions occurred as we expected based on the statechart semantics. In particular, we can verify that the history state is modeled correctly as follows. Before exit transition *T2* fires (Step 4), the currently active nested states of region *MONITORING* of composite state *ON* is *ACTIVE*. After the firing of entry transition *T8* (Step 5), which targets the history state, state *ACTIVE* is activated with two other states (*ON* and *Disconnected*). Moreover, since *ACTIVE* is a composite state, the default state (*Comparing*) associated with *ACTIVE* is also activated.

Experiment 2. The purpose of this experiment is to test if the clearing-history module works correctly. As mentioned earlier, when we model a history state, one issue that we have to face is to clear the old history before we record a new history when an exit transition fires. Note that *T2'* in Fig. 6 has a module 1 that is responsible for clearing history. Module 1 is executed before module 2, which is responsible for deactivating the source states of the exit transition.

To test the history clearing feature, we did the following experiment, based on Experiment 1. We added the following sequence of events, *reset*, *power_off*, *resume* to the event sequence used in Experiment 1. After event *reset* triggers transition *T4*, the currently active state of region *MONITORING* is *Waiting_for_command*. Now, when event *power_off* triggers exit transition *T2*, the new history that would be recorded is state *Waiting_for_command*. So, when event *resume* triggers transition *T8*, state *Waiting_for_command* should be activated.

We set the initial marking of the *IP* place accordingly and ran the net simulation, and recorded the simulation trace. The resulting trace consists of the same five steps as in Experiment 1, but with three additional steps:

```
6. leaving ACTIVE;  trigger: reset;   entering  Waiting_for_command;              // exit transition T4
7. leaving ON;    trigger: power_off;   entering  Off;                            // exit transition T2
8. Off -> ON (H, Disconnected);    trigger: resume;  H -> Waiting_for_command;   // entry transition T8
```

The trace has two steps corresponding to transition *T8*. For the first step (Step 5), *T8* activated state *ACTIVE* because *ACTIVE* was the history recorded for region *MONITORING*. However, for the second step (Step 8), *T8* activated state *Waiting_for_command* since state *Waiting_for_command* was the new history recorded. Thus, this experiment demonstrates that the clearing-history module does clear the old history.

Experiment 3. The purpose of this experiment is to check the logical behavior of our *boundary exit transition* modeling. Consider the following scenario. The system is in its default state *Off*. A sequence of events, *power_on*, *sensor_connected*, *operate*, and *sensor_disconnected*, occurs in that order. After the simulation run using the tool, we obtained the following simulation trace.

```
1. Off -> On (Waiting_for_command, Disconnected,   trigger: power_on;             // entry transitior T1
2. Disconnected -> CONNECTED(IDLE);   trigger: sensor_connected;                  // entry transition T6
3. Idle -> Operating;   trigger: operate;                                         // a nested transition of state CONNECTED
4. leaving CONNECTED;  trigger: sensor_disconnected; entering  Disconnected;      // exit transition T7
```

An examination of the collected simulation trace shows that four transitions have fired and that the final state configuration for the object is (*ON*, *Waiting_for_command*, *Disconnected*). In particular, event *sensor_disconnected* triggers boundary exit transition *T7* (Step 4), and causes the object to leave composite state *CONNECTED* and enter state *Disconnected*.

Experiment 4. The purpose of this experiment is to test the ability of our model to catch a design error in the statechart model. We modified the example of Fig. 4 so that transition *T7* originates from the nested state *Idle* instead of the boundary of state *CONNECTED* (an error type that is not so unlikely to occur in practice). Now, *T7* is a *cross-boundary exit transition*. We modified the target CPN notation accordingly and reran the simulation using the same sequence of events as that of Experiment 3. Now, an examination of the simulation trace showed that only the first three transitions fired and that the final state configuration (set of active states) for the object is {*ON*, *Waiting_for_command*, *CONNECTED*, *operating*}. This revealed a design error because event *sensor_disconnected* is expected to bring the object from state *CONNECTED* to state *Disconnected*. While this experiment was simple, it does illustrate the potential for using the net model to aid automated analysis of the UML diagram.

5.0 Conclusion and future work

In this paper, we focused on presenting a case study to illustrate a transformation approach for formal modeling and analysis of UML statecharts. The case study consists of a set of four experiments. We will continue our research by extending our tool support. Another direction of our work is to provide means and tool support for visualizing and manipulating the simulation traces.

6.0 References

[1] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Fataricza and G. Savoia. Dependability Analysis in the Early Phases of UML-based System Design. *Journal of Computer Systems Science and Engineering,* 16(5), 2001, pp. 265-275.

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

[3] Z. Dong, Y. Fu, and X. He. Deriving Hierarchical Predicate/Transition Nets from Statechart Diagrams. In *Proc Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE'03),* 2003.

[4] H. dels. In *Proc. of Fifth IEEE International Symposium on High Assurance Systems Engineering*, 2000, pp. 83-Gábor and M. István. Quantitative Analysis of Dependability Critical Systems Based on UML Statechart Mo92.

[5] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*, Computing McGraw-Hill, 1998.

[6] Z. Hu and S. M. Shatz. Mapping UML Diagrams to a Petri Net Notation to Exploit Tool Support for System Simulation. In *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE'04),* 2004, pp. 213-219

[7] Z. Hu and S. M. Shatz. Explicit Modeling of Semantics Associated with Composite States in UML Statecharts. Accepted for publication in the *Int'l Journal of Automated Software Engineering*. 2005.

[8] L.M. Kristensen, S. Christensen, K. Jensen, "The Practitioner's Guide to Coloured Petri Nets," *International Journal on Software Tools for Technology Transfer*, 1998, Springer Verlag, pp. 98-132.

[9] W. E. McUmber and B. H. Cheng. UML-based Analysis of Embedded Systems Using a Mapping to VHDL. In *Proc. of IEEE high Assurance Software Engineering (HASE99).* IEEE. 1999.

[10] OMG. UML semantics 1.5, 2003, available at www.uml.org.

[11] R. G. Pettit IV and H. Gomaa. Validation of Dynamic Behavior in UML Using Colored Petri Nets. In S. Kent, A. Evans, and B. Selic, editors, *Proc. of UML'2000 Workshop - Dynamic Behaviour in UML Models: Semantic Questions,* volume 1939 in LNCS. Springer Verlag, 2000.

[12] I. Paltor and J. Lilius. Formalizing UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *UML'99 – The Unified Modeling Language. Beyond the Standard*, volume 1723 in LNCS. Springer, 1999.

[13] J. A. Saldhana, S. M. Shatz, and Z. Hu. Formalization of Object Behavior and Interactions From UML Models. *International Journal of Software Engineering and Knowledge Engineering,* 11(6), 2001, pp. 643-673.

[14] J. Whittle. "Formal Approaches to Systems Analysis Using UML: An Overview." *Journal of Database Management,* 11(4), 2000, pp. 4-13.