Mapping UML Diagrams to a Petri Net Notation for System Simulation

Zhaoxia Hu and Sol M. Shatz
Concurrent Software Systems Laboratory
University of Illinois at Chicago
{zhu, shatz}@cs.uic.edu

Abstract. *UML statecharts are widely used to specify the dynamic behaviours of systems. To support systematic simulation of such models, we propose an approach to map systems specified using UML diagrams to colored Petri net notations. Simulation results are provided in form of self-defined trace files and Message Sequence Charts. A prototype tool is described. One unique feature of our research is the support for user-controlled view of the system simulation.*

1. Introduction

The Object Management Group (OMG) adopted a new paradigm for software development called Model Driven Architecture (MDA) [1] 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 this paper, we propose a UML-CPN transformation framework to introduce dynamic model analysis into UML modeling [2, 3] by mapping UML models to Petri net models, in particular colored Petri nets (CPNs) [4]. This work is aimed at investigating model-driven simulation. In general, simulation can be used to create scenarios based on design models. Evaluation of the scenarios generated by simulation runs helps to reveal potential design errors in an early stage of system development. To leverage on existing techniques and tools, and to formalize UML object models and their interrelationships, we let a net model serve as the engine that drives the simulation. In our framework, statechart diagrams and collaboration diagrams are adopted as our primary notation for modeling behavior. Statechart diagrams are first converted to colored net models; the UML collaboration diagrams are then used to guide the connection of these object models, providing a single CPN for the system under study.

To simulate UML statechart models, a number of commercial statechart simulators are available, such as Rhapsody [5] and ObjectGEODE [6]. Our interest is to investigate techniques that are useful, but not adopted or only weakly adopted, by other tools. We present flexible visualization of scenarios generated through simulation runs to provide users customized views of simulation at different levels of abstraction. In particular, for this paper we investigate techniques for providing meaningful Message Sequence Charts (MSC) [7] that provide views of system simulation to users to help them understand the system itself as well as its properties. We demonstrate techniques for model-driven view control of simulation traces, e.g., selecting some of the model components or a subset of the events.

A prototype tool has been developed to support the automated generation of colored Petri net models from UML notations and to provide users with an interface to control visualizations of the simulation result.

2. Background

We first provide a very brief introduction to statecharts and colored Petri nets.

UML statecharts UML statecharts are an object-based variant of classical (Harel) statecharts [8]. In this paper, we deal with a subset of UML statecharts for the purpose of focusing on the general approach of generating net models from UML specifications and exploring techniques for model simulation. The incremental feature of our approach supports future inclusion of other components of statecharts, such as composite states.

Colored Petri nets Petri nets are a mathematically precise model, and so both the structure and the behaviour of Petri net models can be described using mathematical concepts. We assume that the reader has some familiarity with basic Petri net modeling [9]. Petri nets can be "executed" to perform model analysis and verification. Colored Petri nets (CPNs) [4] are one type of Petri net. In colored Petri nets, tokens are differentiated by *colors*, which are data types. Places are typed by *colorsets*, which specify which type of tokens can be deposited into a certain place. Arcs are associated with inscriptions, which are expressions defined with data values, variables, and functions. Arc inscriptions are used to specify the enabling condition of the associated transition as well as the tokens that are to be consumed or generated by the transition.

This material is based upon work supported by the U.S. Army Research Office under grant number DAAD 19-01-1-0672, and the U.S. National Science Foundation under grant number CCR-9988168.

3. Overview of the UML-CPN architecture

The architecture of the UML-CPN approach is depicted in Fig. 1. The UML-CPN approach integrates three types of application software that are represented by three rectangles in Fig. 1. The Rational Rose tool supports UML modeling, and the Design/CPN tool [10] supports modeling in colored Petri nets. In our framework, we use the Design/CPN tool as the underlying simulation engine. Thus, our approach leverages upon existing theory and tools. The UML-CPN conversion tool is our prototype tool. A typical run of the transformation approach is as follows. The Rational Rose tool is used to design a UML model specified with statecharts and collaboration diagrams. The conversion tool converts the UML model into a CPN model that can then be loaded into the Design/CPN tool for simulation. We call the generated colored Petri net notation supported by Design/CPN the *target model*. The simulation results are presented to the user in two formats: Message Sequence Charts (MSCs) and simulation traces. The simulation traces have a text format and contain complete information regarding the system simulation, while the MSCs can be tailored by the user according to his/her interests regarding the system under study. As we will discuss later, the conversion tool provides the user with an interface to control the views of system behaviour so that only the information that the user is most interested in is displayed in the MSCs.

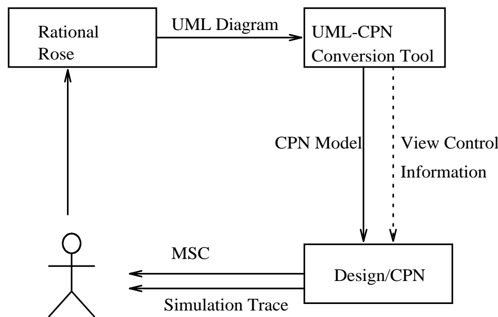


Figure 1. The architecture of the approach

The generation of the target model is based on an abstract net model that was previously presented in [11]. The purpose of defining the abstract model is to be consistent with the ideas of MDA and achieve separation of concerns. Therefore, the transformation of UML specification to colored Petri net notation is divided into two steps. First, a UML model is converted to an abstract net model. In the second step, the abstract model is enriched with the syntax supported by Design/CPN to generate the target model – platform specific model (PSM), in the MDA terminology.

Before we introduce the abstract net model, let us have an overview of the basic mapping between the constructs of UML statecharts and those of colored Petri nets. A statechart consists of states and transitions labelled with events and actions. A Petri net model consists of places, transitions, arcs and tokens. Naturally, the transformation from a statechart to a Petri net is accomplished by the following mappings: a state is mapped to a place; a transition is mapped to a Petri net transition and a set of arcs; and events and actions are mapped to tokens. The concept of "events" is a key factor in defining the execution semantics. In fact the actions of creating, routing, and dispatching of events primarily determine the execution semantics of state machines. Since an event is modeled by a token in the CPN model, we will use *event-tokens* to refer to the tokens derived from events of statecharts from now on.

3.1. Abstract CPN models

An abstract *system-level* model consists of Object Net Models (ONMs) and an Internal Linking Place (*ILP)* place. An ONM, derived from a UML statechart, describes the behaviour of an individual object and defines the token routing mechanism within an object. The *ILP* place defines the communication between the objects.

We start by introducing the structure of Object Net Models (ONMs), as shown in Fig. 2, and described in detail in [11]. An ONM consists of a lifetime behaviour model (*LM*) and a token routing structure. *LM* represents an abstract colored Petri net that is derived from the statechart of an object and describes the object's lifetime behaviour, as defined by the state changes captured in the object's statechart diagram. As shown in Fig. 2, three places – *input place* (*IP)*, *output place* (*OP)*, and *event router place* (*ER)* - and two transitions - *T1* and *T2* - defined the token routing structure for an object. The *input place* of the object holds the event-tokens that will be used by the object. The *output place* of the object holds the event-tokens that will be routed to other objects. The *event router place* holds the event-tokens that are generated by the object. When the object generates an event-token, the token can have a type of either *external* or *internal*. As shown in Fig. 2, the input place, *IP*, is connected to *LM*, indicating that *IP* holds the event-tokens that will be consumed by the object. Thus, for each transition internal to *LM*, if this transition is a triggered transition, there will be an input arc originating from the place *IP*. Likewise, the place *ER* is connected "from" *LM* because *ER* holds the tokens that are generated by the object.

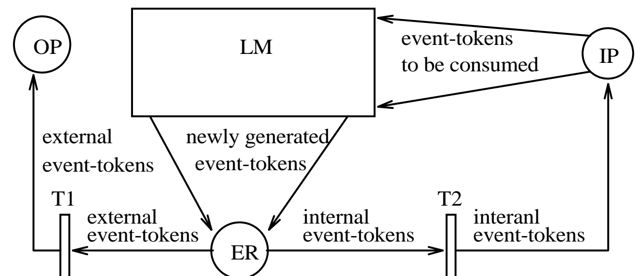


Figure 2. The structure of an Object Net Model

Recall that a state machine is described in terms of a hypothetical machine that has three key components: an event queue, an event processor, and an event dispatcher mechanism [3]. In Fig. 2, the place *IP* holds the event queue. *LM* represents the event processor. The behaviour of the event dispatcher mechanism is handled by the non-deterministic feature of transition firings inherent in Petri net models.

In order to construct a system-level model, the inter-object communications must be integrated into the model. To simplify our approach, we assume the existence of a simple collaboration diagram that defines the event flows between the objects. In the abstract model [11], we defined a special place, an *Internal Linking Place (ILP)* that is used to route event-tokens between the object models.

4. The target model

In order to explore model-driven simulation using an existing simulation engine, we transform the abstract model into a target model, suitable for direct use in the tool called Design/CPN.

4.1. Target model structure

The basic structure for a system-level target model is shown in Fig. 3. A target model consists of modules; in this case, the modules are called pages. Four types of pages are defined for target models: object page, *INL* (Internal Net Linkage) page, main page, and *Init* page. Recall that a system-level abstract model consists of ONMs (one model per object) and an *ILP* place. Accordingly, in the target model, each ONM is represented by an object page that defines the object behaviour described initially with a statechart in the UML notation; and the *ILP* place is represented by an *INL* page that holds the part of the Petri net that is responsible for inter-object communication, which is captured by collaboration diagrams in UML. The net structure of the target model consists of a two-level tree structure and one initialization page, called the *Init* page. The top level of the tree structure is the main page, which contains the high level constructs of the system. The bottom level of the tree structure contains the object pages and the *INL* page that defines the detailed constructs of the system. For convenience, we call object pages and the *INL* page *subpages*. The tree structure alone defines an executable system-level model. Moreover, we define an *Init* page for initializing the net model.

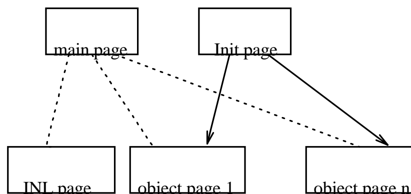


Figure 3. The structure of a target model

We will explain the target model via a simple example. Fig. 4 shows the UML model of a Master-Servant (MS) system where two objects (*Master* and *Servant*) interact with each other. The UML model consists of two statecharts and one collaboration diagram. Initially, the *Master* object is in the state *Init*. When the transition labelled with */Start*, which is a trigger-less transition, fires, a new event *Start* is generated and the *Master* object enters the state *Waiting*. The *Servant* object is initially in the state *Idle*. When event *Start* occurs, the transition labelled with *Start* fires, and the object enters the state *Active*. Fig. 4 (c) shows the simple collaboration diagram that depicts the event flow between the two objects.

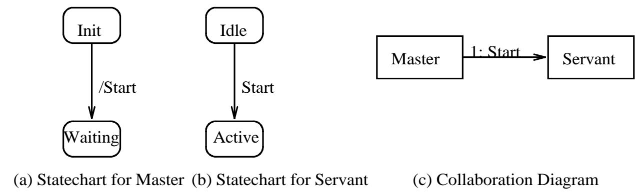


Figure 4. The UML model for the MS system

4.2. Main page

The main page depicts a high-level view of the model structure with the help of substitution transitions. A *substitution transition* represents a *subpage* of the net structure (The effect is the same as if the page that the transition represents appeared physically at the site of the transition). A *subpage* is connected to the main page via special places, which are called *sockets* and *ports*. A *socket* is a place defined on the main page while a *port* is a place defined on a subpage. A socket and a port constitute a pair and they are conceptually the same place. A socket can be associated to multiple ports to connect multiple subpages to the main page. When the model is executed, tokens are allowed to be exchanged through the socket and its associated ports. Fig. 5 shows the main page of the MS example.

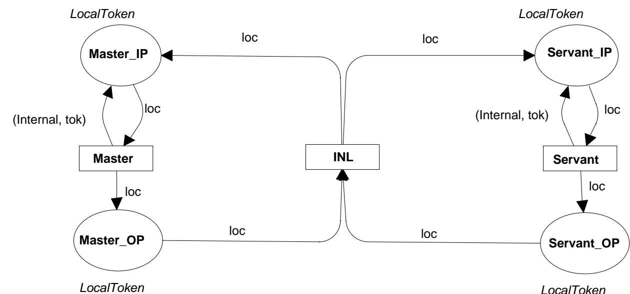


Figure 5. The main page of the MS system

The main page contains three *substitution transitions* that represent, respectively, the *Master* object page, the

Servant object page, and the *INL* page. These subpages represent the bottom level of the tree structure. In Design/CPN, a place is drawn as an ellipse. The four places in Fig. 5 serve as sockets in the model. The sockets connect to the ports in the *INL* page and object pages to "glue" the pages into a system-level net. The postfix "IP" and "OP" of the place names stand for input and output places, respectively. One thing worth noting is that a place has a *colorset* that is a data type that determines the type of tokens that the place is allowed to contain, as we mentioned in Section 2. The four places in the main page have colorset *LocalToken*. Instances of *LocalToken* are event-tokens that we previously introduced in Section 3.

4.3. Implementing ONMs as object pages

An Object Net Model (ONM) describes the behaviour of an object. We refine these models into the target model by mapping each ONM to an object page. Naturally, the structure of an object page captures both the behaviour modeling and the token routing that characterize any ONM.

In order to understand the model behaviour, it is necessary to have in mind the definitions for tokens. For an object page, we define two types of tokens. One type of token represents the activeness of a *state-place*. A *state-place* is a place that is derived from a state of the statechart. We call this type of token *active token*. An active token is denoted as *A*. If a state-place holds an active token, this place is active, denoting that the object is in the state modeled by this place. The other type of token is *event-token* that we introduced previously in Section 3. An event-token can be either an external or internal token, denoted as (External, event_name) or (Internal, event_name), respectively. Fig. 6 shows the object page for the *Servant* object of the MS example.

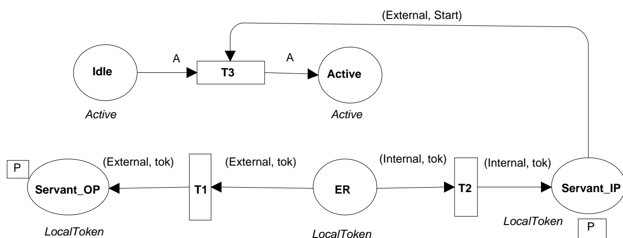


Figure 6. The object page for the Servant object

In Fig. 6, the object behaviour is modeled by the state-places *Idle* and *Active*, and transition *T3*. The other components of the object page implement the token routing structure. The input place *Sevant_IP* and the output place *Servant_OP* place serve as two ports through which the object page is connected to the main page. A port place can be identified by a small rectangular, labelled with a letter *P*. In Fig. 6, assume that the *Idle* place holds an active token, denoting that the object is in the *Idle* state. When an event-token, denoted as (*External*, *Start*), arrives in place *Servant_IP*, transition *T3* is enabled. When *T3* fires, the event-token is consumed, modeling that the event *Start* is dispatched by the event dispatcher of the state machine, and an *active token* is deposited into the place *Active*.

When a UML transition fires, an action can be performed. This type of case is modeled by generating an event-token corresponding to the action. Accordingly, an arc is then added from the corresponding transition to the *ER* (event router) place. Whenever an event-token is created, a type attribute is attached to the token. The attribute can be *Internal* or *External* – specifying if the event is to be responded to by the same object that creates the event (i.e., the creator object) or if the event is to be sent to other objects.

Due to lack of space, details on the *INL* and *Init* pages are not provided.

5. Simulation traces and visualization

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, and newly generated event. Fig. 8 (a) shows a very simple example simulation trace of the MS system. It records the history for firing two transitions.

Another method of observing simulation results is by using Message Sequence Charts [7], which have an intuitive graphical appearance and capture the message passing between the objects of a system. An MSC shows a history of events in terms of a timeline for each object. We define code segments that invoke an MSC library [12] to create the visual MSC diagrams. The MSC library provides two functions, call them *c1* and *c2*, for generating components of MSCs. Function *c1*, which has two parameters – an object and a label – generates a solid square on the timeline associated with the object. The label-parameter is placed next to the square icon. We invoke function *c1* with an object-parameter *obj* and a label-parameter *event_name* to visualize that a triggering event, *event_name*, is consumed by object *obj*. Function *c2*, which has three parameters – a sender object, a receiver object, and a label – generates a labelled arrow originating from the timeline of the sender object and targeting the timeline of the receiver object. We invoke function *c2* with the parameters *sender_obj*, *receiver_obj*, and *event_name*, to visualize that an event with name

event_name is sent from object *sender_obj* to object *receiver_obj*.

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 Message Sequence Charts. We call these transitions *critical transitions*. For example, transition *T3* in Fig. 6 is a critical transition. Transition *T3* is directly derived from the transition of the statechart in Fig. 4 (b). On the other hand, transitions *T1* and *T2* belong to the second category of net transitions; they are support transitions. In this case, *T1* and *T2* are defined for routing event-tokens.

The main components of Message Sequence Charts are those that visualize the message passing between objects. The following procedure outlines the basic steps for defining the code segments to generate these components with respect to some object model, *obj*.

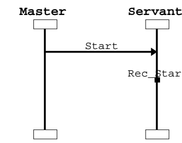
Notations
- *CT(obj)*: The set of critical transitions associated with the target model for object *obj*.
- *consume(t, e)*: A Boolean condition that evaluates to true if transition *t* is specified to consume an event-token *e*.
- *generate(t, receiver_obj, e)*: A Boolean condition that evaluates to true if transition *t* is specified to generate an event-token *e* that can be consumed by a critical transition associated with object *receiver_obj*.

```
foreach t ∈ CT(obj) do
    if consume(t, e1) and generate(t, receiver_obj, e2)
    then generate a code segment that contains the
            following two function calls:
              c1(obj, e1);
              c2(obj, receiver_obj, e2);
    else if consume(t, e1)
        then generate a code segment that contains the
                following function call:
                  c1(obj, e1);
         else if generate(t, receiver_obj, e2)
             then generate a code segment that contains
                     the following function call:
                       c2(obj, receiver_obj, e2);
             endif
         endif
    endif
enddo
```

When a critical transition fires the associated code segment is executed, which invokes the MSC library functions. Thus, the following information is captured in an MSC: consuming of the triggering event, and sending of newly generated events (if there are any) to destination objects. For instance, consider Fig. 8 (b). The arrow represents that the *Master* object generates a new event called *Start* and sends this new event to the *Servant* object. The solid square represents that the *Servant* object has received the event *Start* and this event triggers a transition.

Master:
  Init -> Waiting
  | Start
Servant:
  Idle -> Active
  Start |

(a)



(b)

Figure 8. A simulation trace and MSC for MS system

5.1. User-controlled views of system simulation

A complex distributed system may consist of many objects that communicate with each other through message passing. As a means to control the complexity of systems analysis, designers view systems at different levels of abstraction. To aid this process, we want a designer to be able to reason about the behaviour of a subset of the objects or the occurrences of some particular events. Accordingly, an MSC can be defined to capture the behaviour of a subset of the objects and/or the occurrences of some selected events. In this section, we introduce the idea of filters to tailor the views for the system behaviour. We have defined two types of filters: *object filters* and *event filters*. Object filtering allows the user to select objects of interest – those objects whose behaviours are to be captured in the MSCs; and event filtering allows the user to constrain the information displayed in the MSCs by selecting events of interest. These two types of filters can be used together to control the views of system behaviour. Our prototype tool provides interfaces to allow the user to choose different views of the system behaviour using these filters.

The key idea for implementing the view control is that the target model is parameterized based on the selected objects and events so that the model can produce different views during the simulation, according to the user's choices. The view control technique can help a user check for the following types of properties (among others) in simulation traces: A given event occurs; an event does not occur; an event *P* is followed/preceded by another event *Q* (we will illustrate this last one in our example).

6. An example

We illustrate our approach by a small example of a gas station system. This example is adapted from [13]. The system consists of a customer and a pump that processes the customer's request for filling gas. The customer must prepay for the gas. After the customer pays, (s)he is allowed to select the gas grade and then press the nozzle. Then the pump starts to fill gas into the tank. The pump stops filling when the prepaid money is spent or the tank is full. If the customer's prepaid amount is more than the value of the gas filled into the customer's gas tank, the

pump will prompt the customer to pick up the change. The customer can change his/her decision and cancel the request for filling the tank after (s)he pays or selects the gas grade. After the customer cancels the request, the pump will return the customer's prepaid money. A simple UML model for this example consists of two statecharts and one collaboration diagram. Statecharts for the *Customer* object and *Pump* object are shown in Figures 9-10[1]. As we will discuss shortly, these models contain some error to be revealed by our simulation analysis.

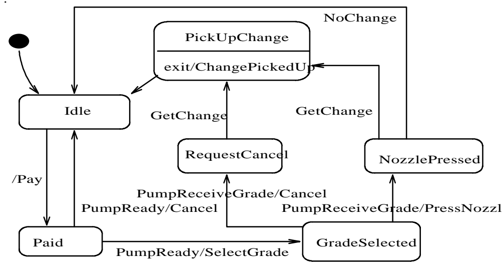


Figure 9. Statechart for Customer

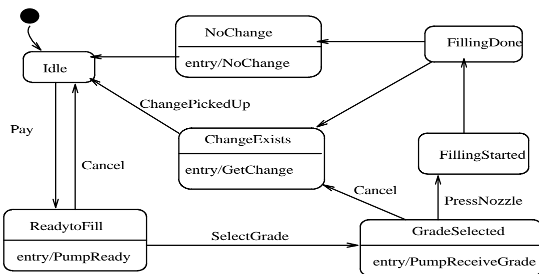


Figure 10. Statechart for Pump

For illustration, let us suppose that we want to check the following property:

Once the customer cancels the request for purchasing gas, the customer's prepaid amount should be returned.

This property can be checked by inspecting the MSCs generated from simulation runs. Fig. 11 (a) shows an MSC generated by a simulation run using our tool. We can see from the figure that many messages have been passed between these two objects. Since we are particularly interested in two events, *Cancel* and *GetChange*, we use event filtering to help us remove unwanted information from MSCs. We use the interface provided by our prototype tool to select the two events of interest. Now, the MSC generated from a simulation run is shown in Fig. 11 (b). We can clearly identify that none of the first three occurrences of *Cancel* is followed by an occurrence of *GetChange*, indicating an error in terms of the desired behaviour of the source model. Although this is a very simple example, it illustrates the intended purpose of our model-driven view control. Due to lack of space, we let the reader determine how to modify the source statechart models.

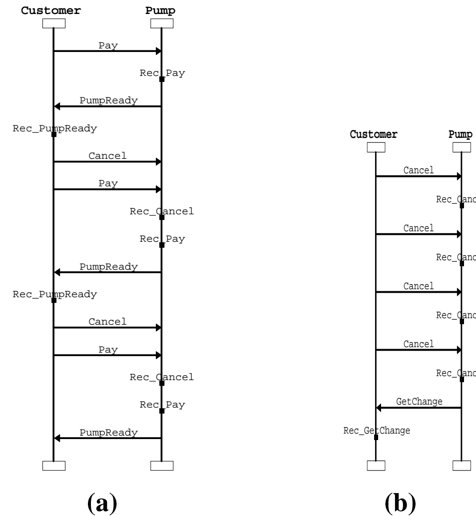


Figure 11. Message Sequence Charts for Gas Station

7. Related work

There are other research efforts that concentrate on supporting validation and analysis of UML statecharts by mapping UML diagrams to other formal target notations [14]. One such research effort aimed at validating UML models was the work on a tool called vUML [15]. This work used the information contained in the class diagrams, statechart diagrams and collaboration diagrams of a model to generate the Promela specification. The SPIN model checker was used to perform the verification. In [16], the authors present a branching time model-checking approach to the automatic verification of correctness of UML statecharts. In this work, statecharts are first translated into hierarchical automata to provide a formal semantic basis for verification. In our work, we adopt Petri nets as the target language since Petri nets have strength in their graphical notations and a mature theoretical base. Furthermore, as highlighted in this paper, we have found that net models can be flexibly adapted to drive simulation experiments.

Work with a similar motivation as ours includes [17]. Our work differs from this work in that we hide the underlying simulation engine from the end user by presenting the simulation results using the constructs from

[1] In the statecharts, there exist transitions that originate from the same source state and have the same triggering event (or, both transitions do not have a triggering event). This is being done to model a pure nondeterministic choice.

the original UML model. In addition, our transformation approach is automated. The work of Dong et. al. [18] is closely related to our work. They presented an approach of using Hierarchical Predicate Transition Nets (HPrTNs) to define and integrate UML statechart diagrams and collaboration diagrams. Thus, these efforts are quite complementary to our basic approach. However, the HPrTN model is defined in a more abstract manner than our CPN model. Also, our research extends the basic translation technique to create a model that can be imported into an existing CPN support tool and to use this tool for investigating model-driven simulation.

Other research efforts that relate to statechart simulation include those presented in [19, 20]. One idea that separates our work from other statechart simulation research is that we introduce view control into the simulation process to help a user understand and reason about the behaviour of a system.

8. Conclusions and future directions

In order to explore the validation and verification of UML models through transformation, we follow a transformation framework to structure UML models as colored Petri net models. We enrich the abstract net model and present a pragmatic CPN model so that the resulting model can be imported into Design/CPN for simulation and analysis. The transformation is based upon the execution semantics of state machines. To help users to facilitate the simulation features supported by Design/CPN, we derive Message Sequence Charts to visualize simulation results. A prototype tool has been developed to support the automated generation of executable CPN models from UML notations. One unique feature of the tool, which is facilitated by the net construction, is the support for interfaces that enable a user to control the visualization of system simulations. We provide object filters as well as event filters to control the views for the system behaviour.

One direction for future work can be to extend our methodology so that it supports the transformation of more complex features of UML statecharts, such as concurrent composite states. Another direction is to investigate the integration of other UML diagrams in our approach to strengthen the behavioural modeling and analysis.

Acknowledgments

We thank C. Xiong for her help in development of the prototype tool. We also thank the reviewers for their helpful suggestions that improved the presentation of the work.

References

[1] OMG. Model Driven Architecture, www.omg.org/mda.

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

[3] OMG. UML semantics 1.5, 2003. www.cgi.omg.org/uml.

[4] 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.

[5] E. Gery, D. Harel, and E. Palatshy. “Rhapsody: A Complete Lifecycle Model-Based Development System.” In *Proc. Int’l Conf. on Integrated Formal Methods*, 2002. pp. 1-10.

[6] ObjectGEODE. www.csverilog.com.

[7] ITU-T Recommendation Z.120: Message Sequence Chart. International Telecommunication Union; Telecommunication Standardization Sector (ITU-T), 1999.

[8] D. Harel. “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming*, vol. 8, 1987, pp. 231-274.

[9] T. Murata, “Petri Nets: Properties Analysis and Applications.” In *Proc. the IEEE*, 77(4), April 1989, pp. 541-580.

[10] Design/CPN, www.daimi.aau.dk/designCPN/.

[11] J. A. Saldhana, S. M. Shatz, and Z. Hu, “Formalization of Object Behavior and Interactions From UML Models.” *Int’l Journal of Software Eng. and Knowledge Eng.,* 11(6), 2001, pp. 643-673.

[12] MSC Library, www.daimi.au.dk/designCPN/libs/mscharts/

[13] Gas station example, sunset.usc.edu/classes/cs599_2000/GasStation.pdf

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

[15] J. Lilius and I. Paltor, “vUML: A Tool for Verifying UML Models.” In *Proc. the 14th Int’l Conf. on Automated Software Engineering*, IEEE. 1999, pp. 255-258.

[16] S. Gnesi, D. Latella, and M. Massink. “Model checking UML statechart diagrams using JACK” In *Proc. Int’l Symp. on High Assurance Systems Eng.* IEEE. 1999, pp. 46-55.

[17] R. G. Petti IV and H. Gomaa, “Improving the Reliability of Concurrent Object-Oriented Software Designs.” In *Proc. the 9th Int’l Workshop on Object-oriented Real-time and Dependable Systems (WORDS),* October, 2003.

[18] Z. Dong, Y. Fu, and X. He. “Deriving Hierarchical Predicate/Transition Nets from Statechart Diagrams.” In *Proc. Software Eng. and Knowledge Eng.,*2003, pp.150-157

[19] A. Egyed and D. Wile, “Statechart Simulator for Modeling Architectural Dynamics.” In *Proc. the 2nd IEEE/IFIP Conf. on Software Architecture (WICSA)*, 2001, pp. 87-96.

[20] I. Ober, S. Graf, and I. Ober, “Validating Timed UML Models by Simulation and Verification.” *UML´2003 Workshop – Specification and Validation of UML Models for Real Time and Embedded Systems,* 2003.