Simulation-Based Analysis of UML Statechart Diagrams: Methods and Case Studies[1]

Jiexin Lian, Zhaoxia Hu and Sol M. Shatz
Department of Computer Science
University of Illinois at Chicago
Chicago, IL, U.S.A.

Abstract - We present a comprehensive UML statechart diagram analysis framework. This framework allows one to progressively perform different analysis operations to analyze UML statechart diagrams at different levels of model complexity. The analysis operations supported by the framework are based on analyzing Petri net models converted from UML statechart diagrams using a previously proposed transformation approach. After introducing the general framework, the paper emphasizes two simulation-based analysis operations from the framework: direct MSC inspection, which provides a visual representation of system behavior described by statechart diagrams; and a pattern-based trace query technique, which can be used to define and query system properties. Two case-study examples are presented with different emphasis. The gas station example is a simple multi-object system used to demonstrate both the visual and query-based analysis operations. The early warning system example uses only one object, but features composite states and includes analysis specifically aimed at one composite state feature, history states.

Keywords: Design analysis, UML statecharts, Formal methods, Petri nets

1. 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) [3]. 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 [32].

We previously proposed a transformation approach to introduce dynamic model analysis into UML modeling [15, 31]. The approach leverages the ability to map UML models – in particular, statechart diagrams – to Petri net models. The Petri net models are represented using colored Petri nets (CPNs) [18]. Statechart diagrams are converted to colored Petri net models with the intent that a standard CPN analyzer can be used to support assorted analysis of the resulting net model. We provided methodologies to exploit the simulation features supported by a colored Petri net tool, Design/CPN [6], and a prototype UML-CPN

[1] This material is based upon work supported by the U. S. Army Research Office under grant number W911NF-05-1-0573.

conversion tool was created. Two formats of simulation reports were considered in this earlier work [15]. One is a simple textual notation that directly captures the objects, states, and events of simulation traces. The other is a graphical notation based on Message Sequence Charts (MSCs). The MSC has an intuitive graphical appearance and is used to visualize the interaction among objects. MSCs can be tailored by the user according to his/her interest regarding the system under study.

While the previously proposed UML-CPN transformation approach establishes a foundation for exploiting the mature theory and tools for Petri nets, there is still a gap between statechart diagram development and analysis. Different types of analysis are needed at different development stages, and different forms of analysis imply varying degrees of complexity. To most effectively bridge this gap, there are two challenges. First, as a starting point, it is desirable to define a *flexible* statechart analysis methodology – one that can support a range of analysis techniques within a common framework. Secondly, one needs specific analysis techniques that fit into the overall framework and can provide feedback on the behavior that is inherent to a statechart-based design. Furthermore, we believe that both visual-based feedback and query-based feedback are important goals to achieve.

In this paper, we tackle both challenges mentioned previously. To address the first challenge, we present a comprehensive statechart diagram analysis framework. The framework takes advantage of the abundant CPN-based analysis tools to build a flexible, "multi-modal" analysis procedure. We refer to the framework as "multi-modal" since it supports a range of analysis techniques. By inheriting the conversion facility of our earlier UML-CPN approach, the analysis framework still begins with the conversion of a statechart diagram into a CPN. Then the framework provides the UML designer a set of different analysis operations: direct MSC inspection, pattern-based trace query, and model checking. These operations imply different degrees of analysis intensity. Depending on the complexity of the statechart diagram, the UML designer can progressively perform a set of analysis operations, where the analysis typically starts from the least intensive operation, using a relatively light-weight analysis method. If needed, the designer can then choose to initiate a deeper form of analysis using a more heavy-weight analysis method. In this way, our framework directly supports the concept of flexible analysis for UML statecharts.

To address the second challenge, we emphasize two simulation-based analysis operations from the framework: direct MSC inspection, which provides visual representation of system behavior described by the statechart diagram; and a pattern-based trace query technique, which can be used to define and query system properties. Since the heart of this paper deals with simulation-based analysis, it is important to note that analysis results are based solely on simulation traces, which means that the conclusions are only valid with respect to those traces.[2] Thus, simulation-based analysis cannot provide results as strong as

[2] In the same way that testing results are only valid with respect to the set of test cases used to establish the testing results.

formal verification (over a full state-space). Still, simulation is widely recognized as a valuable analysis method, which can help identify design errors without expensive state-space analysis, even if it does not prove model correctness.

Statecharts diagrams are just one part of UML. Although UML can describe models with assorted properties, we focus our research on models of objects with significant state-based behavior. Also, we consider communication at the statechart level to be asynchronous and based on signal-events. While statecharts do allow for synchronous communication using call-events, we have currently limited our attention to the simpler signal-events since they are commonly accepted and appear to be much more often used in practice. Finally, the statechart diagrams discussed in this paper follow the semantics defined for UML 1.5, since this was the UML standard at the time the model transformation research was done (i.e., translation to Petri net notation). But, since OMG has upgraded UML to version 2.0, in Section 2 we will discuss the impact of UML 2.0 on our transformation approach. The key observation is that the presented framework retains its core value if applied to UML 2.0.

We also point out that Design/CPN has recently been replaced by a new software toolkit called CPN Tools. Although CPN Tools provides significant enhancements in terms of the user interface, it mainly inherits the analysis methodology and file format from Design/CPN. Since our interest is in net analysis, not manual net model creation (drawing), the enhancements associated with CPN Tools are not very valuable to our work. In addition, CPN Tools does not yet support some capabilities useful to our analysis framework, e.g., the MSC library [21]. In the future, it should be conceptually easy to replace Design/CPN by CPN Tools in our analysis framework.

It should be noted that there do exist a number of commercial and academic tools to analyze statechart diagrams by simulation or formal model checking. For example, Rhapsody [10], ObjectGEODE [23], and Rational Rose RT [30] can simulate statecharts directly, while vUML [19] performs model checking to verify statechart models. However, these techniques are not designed to explicitly support a broad range of analysis operations that can be progressively performed to deal with different levels of model complexity. Furthermore, in contrast to direct simulation of statechart diagrams, our analysis framework benefits from the fact that we convert statecharts to the CPN model, which serves as an intermediate model that can be automatically manipulated to customize a simulation run. It is the existence of the underlying CPN model that allows us to perform flexible simulation control without modifying a statechart diagram directly. This type of simulation control – realized by a concept called filtering, which will be presented in Section 3 – is not featured in other techniques that do simulation directly on the source statechart model.

The rest of the paper is organized as follows. Section 2 provides background information. This begins with a summary of some key translation concepts, previously published. The impact of UML 2.0 on our technique is also discussed in this section. Section 3 outlines our UML statechart analysis framework and discusses two key simulation-based analysis methods. Section 4 presents two case-study examples, the gas station case and the early warning system case. Finally, Section 5 provides a conclusion and mentions future work.

2. Background

As we already discussed, there do exist other tools for statechart simulation (e.g., Rhapsody [10]). However, unlike our approach these tools are not designed to explicitly support a broad range of analysis operations. Furthermore, our analysis framework benefits from the fact that we convert statecharts to the CPN model, which serves as an intermediate model and opens up opportunities to exploit existing theory and tools. Therefore, we focus our attention on prior work related to UML statecharts and CPN modeling. In particular, the rest of this section provides background information on UML statecharts, Colored Petri nets, and conversion from UML statecharts to CPN.

For the most part, details on our previously published conversion technique [13, 14, 15, 31] are irrelevant to understanding the analysis framework itself, which is the focus of this paper. As mentioned before, we are using the semantics of statecharts associated with UML 1.5. Since UML 1.5 statecharts have evolved into UML 2.0 state machines, we will further discuss the impact of this evolution in Section 2.3. It is also worth noting that conversion of a semi-formal statechart model to a formal Petri net model requires semantic interpretation of the semi-formal model. The main difficulty comes with complex semantics that are associated with composite states. From a model transformation perspective, this has been addressed in a previous paper [14].

2.1 UML Statecharts and Colored Petri nets

UML statecharts are an object-based variant of classical (Harel) statecharts [12]. A statechart contains states and transitions. Statecharts extend finite state machines with composite states to facilitate describing highly complex behaviors. The execution semantics of a state machine [24] is described in terms of a hypothetical machine whose key components are: 1) An event queue that holds incoming event instances until they are dispatched; 2) An event dispatcher mechanism that selects and de-queues event instances from the event queue for processing; and 3) An event processor that processes dispatched event instances. The execution semantic of state machines provides the basis for the transformation from UML statecharts to colored Petri net models.

Petri net models are mathematically precise models, and so both the structure and the behavior of Petri net models can be described using mathematical concepts. We assume that the reader has some familiarity with basic Petri nets [22], but we can start with a general reminder of Petri net concepts. By mathematical definition, a Petri net is a bipartite, directed graph consisting of two types of nodes, i.e., places and transitions, and a set of arcs, supplemented with a distribution of tokens in places. An arc connects a transition to a place or a place to a transition. The distribution of tokens among places at certain time defines the current state of the modeled system. Transitions are enabled to fire when certain conditions are satisfied, resulting in a change of token distribution for places. With its formal representation and well-defined syntax and semantics, Petri nets can be "executed" to perform model analysis and verification.

Colored Petri nets (CPNs) [17] are one type of Petri net model. 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.

2.2 Converting UML Statecharts to Colored Petri Nets

There are a number of related efforts in the area of modeling to support validation and analysis of UML statecharts [1, 2, 7, 9, 20, 26, 27, 28, 29]. One closely related line of research is the work of Pettit, et al. [27, 28, 29]. They also consider conversion from UML statechart diagrams to CPN, with Design/CPN being used to analyze the generated CPN model. However, since their work is part of an effort to provide a systematic and seamless integration of CPNs with object-oriented software architectures, their conversion is implemented in the flavor of facilitating such integration. In contrast, our conversion technique aims to define universal conversion rules, so that the conversion can be automated in support of a stand-alone analysis capability. Another closely related work is proposed by Baresi and Pezze [1], but our work goes further in terms of defining techniques and tools to assist with the analysis of generated net models. In [14] we provide details about the relationship between our general conversion approach and other efforts. We do not discuss those issues in detail here since the focus of this paper is on comprehensive, multi-modal analysis and associated case studies, rather than the conversion process.

We summarize some of the key translation concepts that we have previously defined [14, 15, 31]. The transformation of UML statechart to colored Petri net notation is divided into two phases. First, UML statecharts are converted to net models, one net model for each object. Then, collaboration diagrams are

used to connect the object net models into a system-level model. The process for connecting object net models to create the system-level model is outside the scope of this paper, but is discussed in [31].

A statechart consists of states and transitions labeled with events and actions. A Petri net model consists of places, transitions, arcs and tokens. In a fairly natural way, the conversion of 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. Composite states complicate the generation of net models because the semantics associated with composite states are often not explicitly observed in statecharts. To simplify the transformation process for composite states, we introduced an intermediate model [14] 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. The rest of this section highlights the basic procedure for converting composite states into net models to provide the background for the second case study in Section 4.

2.2.1 An Intermediate Model for Composite States

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. Here, we briefly refer to a few key features associated with composite states (entry transitions, exit transitions, and history states). Full details on modeling composite states are given in [14].

Entry transitions: An entry transition is a transition that leads directly to a composite state. When an entry transition fires, the composite state is activated. The basic form of the intermediate model for entry transitions of a composite state is a fork transition as shown in Fig. 2.1. 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* [14]. A boundary entry transition targets a composite state while a cross-boundary entry transition targets nested states of a composite state.

Exit transitions: An exit transition is a transition that emanates from a composite state. As a result of firing an exit transition, a composite state is exited. The basic form of the intermediate model for an exit transition is a set of transitions as shown in Fig. 2.2. 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 [14]. Control states are depicted as ovals in our intermediate models, as shown in Fig. 2.2.

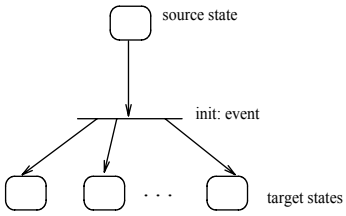


Fig. 2.1 The intermediate model for entry transitions

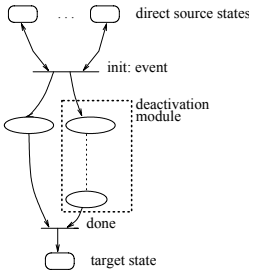


Fig. 2.2 The intermediate model for exit transitions

History states: A history state allows a composite state that contains sequential substates to remember the most recently active substate prior to exiting from the composite state. 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 resetting 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 two-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.

2.2.2 Target Model Generation and Simulation for Composite States

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 that handles selection and dispatching of an event-token in the event queue; makes the dispatched event-token, also called the *current* event-token, available to net transitions that may or may not be associated with composite states; and enforces transition priorities. This event dispatching model is discussed in [14].
   - 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 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.

2.3 Impact of UML 2.0

UML 2.0 represents a major revision to the UML standard [25]. Although UML 1.5 statecharts and UML 2.0 state machines mainly use the same notions to describe state transitions, UML 2.0 explicitly introduces *Structured Class*, where the state machine is a modularized sub-machine of the class. Although UML 1.5 does not address the relationship between a class and a statechart diagram, our conversion technique has already been defined so that each object to be analyzed is represented by a statechart. From this perspective, our conversion technique is compatible with the UML 2.0 standard.

Multiple entry and exit points are now legal in UML 2.0. This allows different ways to enter or leave a state machine diagram. In contrast, our conversion technique converts all the events to CPN event tokens. For each object, there is a place *IP* that serves as the common destination place for all incoming event tokens, and a place *OP* that serves as the common source place for all the out-going event tokens. So it would be natural to revise our technique so that *IP* and *OP* can be divided into several different *IP* and *OP* place nodes. Then each entry point can be converted to a specific *IP*, while each exit point can be converted to a specific *OP*.

Besides the extensions mentioned above, “The syntax and semantics of UML state machines have remained reasonably consistent throughout UML's history, although there are occasionally minor

modifications" [5]. Other improvements, including protocol state machines, redefinition, notation enhancement (action blocks and state lists), are not concerned with the semantic aspects, but provide enhancements for the model development procedure. For example, UML 2.0 also formalized and included the concept of a *protocol state machine diagram* as part of the specification. In contrast, in UML 1.5 it was already common practice to use a state machine diagram to create a protocol state machine diagram, which specifies the legal sequence of events. Since our conversion technique focuses on the transformation of UML statechart semantics, the enhancements found in UML 2.0 and the resulting case tools are mostly unimportant for our work. Further details of adapting our conversion technique to the UML 2.0 standard are beyond the scope of this paper. As a final comment, we note that Crane and Dingel have observed that the existence of the semantic differences between UML statecharts, classical Harel statecharts and Rhapsody statecharts may impede the transformation between these three most popular statechart formalisms [5]. For this research, we are basing our technique only on the UML statechart formalism since it is widely studied and used in both research and industry.

3. Comprehensive UML Statechart Diagram Analysis Framework

The core components of the comprehensive UML statechart diagram analysis framework are the UML-CPN conversion component and three analysis operations. The three operations are: direct MSC inspection, which means browsing the visualized simulation results manually, pattern-based trace query on the simulation traces, and CPN-based model checking. Fig. 3.1 shows the architecture of our UML statechart diagram analysis framework.

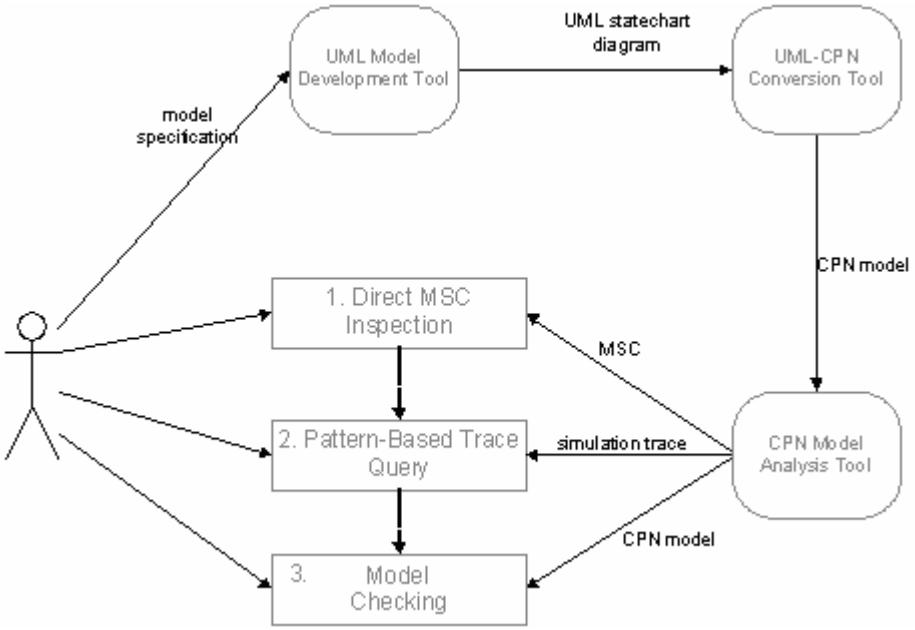


Fig. 3.1 UML statechart diagram analysis framework

As shown in Fig. 3.1, a UML statechart diagram analysis procedure starts from model development. Then the developed UML statechart diagram is converted to a CPN model. After that, the three different analysis operations will be progressively performed on the resulting CPN model, until the analysis need is satisfied. Note that different analysis operations may require different pre-defined CPN model processing to generate input for analysis. For example, direct MSC operation needs the resulting CPN model to be processed for MSCs generation, while pattern-based trace query requires trace files from the pre-defined trace file generation process. The dotted arrows in Fig. 3.1 indicate the typical order of application for the analysis operations – from least intense to most intense, or from light-weight analysis to heavy-weight analysis.

For both direct MSC inspection and pattern-based trace query, the size of traces (MSCs can be seen as visual traces) is directly relevant for analysis accuracy. Generating a complete set of traces is generally infeasible and individual traces can be of infinite size. Even in the case of finite traces, one must make a trade-off between analysis accuracy and the practical problem of limiting the size of trace sequences. While this problem of "how much simulation is enough" is clearly important, it is beyond the scope of this paper, although we do provide data on the number of simulation traces and the length of traces used for the case study experiments in Section 4. Finally, our simulation traces are analysis-independent. We first collect traces and then perform analysis over these traces. This allows multiple queries to be evaluated on the same simulation traces, but also requires careful construction of queries since a query may be intended to only apply to some limited portions of an arbitrary trace sequence. The property pattern notation we adopt (Section 3.2) for expressing queries provides a *scope* construct explicitly for this purpose.

Direct MSC inspection is the most basic of the simulation-based analysis operations. By directly inspecting a complete or tailored version of a MSC generated by a simulation run of a CPN model analysis tool (Design/CPN in this paper), the UML designer can abstract useful information for a simple statechart diagram. But, the MSC associated with a complex source model may contain too much information, making direct MSC inspection cumbersome or impractical. The UML designer then can perform the second operation, pattern-based trace simulation. This supports query-based reasoning about behavioral characteristics of simulation traces. But, since simulation traces only represent a sample of all possible system behaviors, true behavior verification is not possible. To achieve this degree of analysis, the designer can employ the third operation, model checking. So, model checking complements and extends the analysis capability of the pattern-based trace method by supporting verification, but at the significant increase in cost due to the need to cope with state-space complexity. Model checking can be

applied to the CPN model by either using a CPN-specific tool that supports model checking or by using a more generic model checker. As noted earlier, others have studied the idea of model checking applied to statecharts. Thus, we do not discuss this topic further. It should be obvious that such methods, especially those already existing in Petri net tools, can be "plugged-into" our analysis framework. We do note that we have previously demonstrated how Petri-net based model checking can be used to support property verification for a type of agent model [33]. Thus, it is reasonable to expect that a similar approach can be applied to statechart diagrams using the target models associated with the work presented in this paper.

Of course a designer/analyst can choose to employ any or all of the three basic operations, depending on the scope of the analysis challenge. For example, if a designer knows that direct MSC inspection will not be helpful for his/her need, the analysis can start directly from pattern-based trace query. In this sense, our framework provides some degree of flexibility.

3.1 Direct MSC Inspection

Direct MSC inspection means browsing the visualized simulation result manually. Although this is the most basic analysis method, direct MSC inspection is the least intensive among all the three operations and can be effective for simple statechart diagrams.

The Message Sequence Chart (MSC) [16] is used to visualize the simulation result during a simulation run. It has an intuitive graphical appearance and is very useful in capturing normal behavior of systems, or to discuss pathological cases of interaction between components. An MSC shows a history of events in terms of a timeline for each object in the model. In our MSCs, a horizontal arrow labeled with an event name denotes the sending and the reception of the event from one object to another, which makes the event available to the target object. A solid rectangle labeled with an event name denotes that the event is consumed, i.e., it triggers a transition.

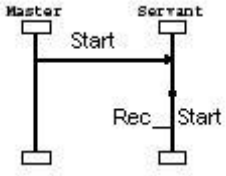


Fig. 3.2 A MSC for the Master-Servant system

Fig. 3.2 shows an example MSC for a Master-Servant system where two objects (*Master* and *Servant*) interact with each other. The horizontal arrow represents that the *Master* object generates a new event called *Start* and sends this new event to the *Servant* object. The solid rectangle represents that the event *Start* triggers a transition of the *Servant* object. We use an ML library [21] to create MSCs, which

are controlled by the simulation of a colored Petri net. Further details on the implementation of MSCs is discussed in [15].

A major limitation of direct MSC inspection is scalability. 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, a designer should be able to reason about the behavior of a subset of the objects or the occurrences of some particular events. Accordingly, an MSC can be defined to capture the behavior of a subset of the objects and/or the occurrences of some selected events. Depending on what objects and events are selected, the MSC can then provide different views for the behavior of the system. We apply the idea of filters to tailor the views for the system behavior. Two types of filters are defined and developed: *object filters* and *event filters*. These two types of filters can be used together to control the views of system behavior.

Object filtering allows the user to select objects of interest – those objects whose behaviors are to be captured in the MSCs. Furthermore, we provide two refinements of object filters, one stricter than the other in terms of the information displayed in the MSC. We call the first *constrained filter* and the second *relaxed filter*. In order to distinguish the two types of filters, we examine what is involved in capturing object behavior. An object changes its state via firing of transitions. Typically, two steps are involved in the firing of a transition. First, the object receives the triggering event and the transition is enabled and fires; so that the triggering event is consumed. Second, the object can generate new events that are to be sent to other objects. Depending on which object filter is chosen, one or both steps of a transition-firing may be filtered, and thus not captured in the MSC. To be more specific, let us consider one object. A triggered transition of this object fires and generates a new event, which will be sent to other objects. With the *constrained filter*, if the sender object corresponding to the triggering event is not currently selected as an object of interest, the consumption of the triggering event is filtered - the solid rectangle, specifying the consumption of the event, will not be shown in the MSC. Also, if some receiver object of the newly generated event is not currently selected as an object of interest, the horizontal arrow, specifying the sending and reception of this event to this object is filtered. However, with the *relaxed filter*, the behavior of a selected object will be completely captured, regardless of which other objects are selected.

In general, the *constrained filter* provides a view illustrating the passing of messages only among the selected objects while the *relaxed filter* provides the complete behavior of the selected objects. Fig. 3.3-3.4 show the system views for an example microwave oven system [31] when a constrained filter and a relaxed filter are applied, respectively. The microwave oven system has four objects: *User*, *Oven*, *LightTube* and *PowerTube*. Each object can send and receive some events. In both the constrained filter

version (Fig. 3.3) and the relaxed filter version (Fig. 3.4), *User* and *Oven* are selected, while *LightTube* and *PowerTube* are filtered. As we can see, event *TurnOn* and the consumption action *Rec_LightOn* are filtered in Fig. 3.3 because the receiver object of event *TurnOn* and the sender object of event *LightOn* are both filtered. In contrast, *TurnOn* and *Rec_LightOn* are retained in Fig 3.4 to show a complete view for object *User* and *Oven*.

The second type of filtering, event filtering, allows the user to constrain the information displayed in the MSCs by selecting events of interest. If an event is not selected, no information associated with this event will be displayed in the MSC.

We have implemented both types of object filters and the event filter in the UML-CPN conversion tool as conversion options. By default, all objects and events are considered to be selected in the UML-CPN conversion tool, and simulating the resulting CPN model will yield the complete version of MSC, where no horizontal arrows or solid rectangles (with associated events) will be filtered.

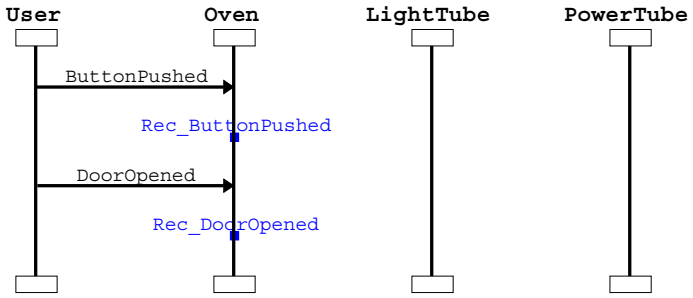


Fig. 3.3 A system view with a constrained filter (the User and Oven objects are selected)

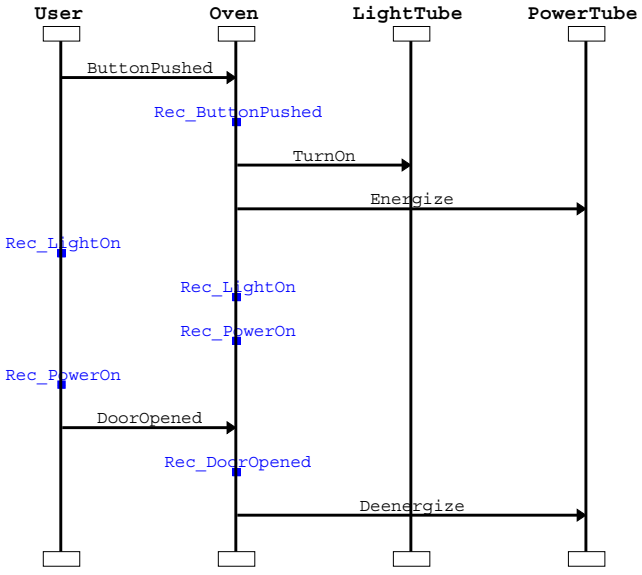


Fig. 3.4 A system view with a relaxed filter (the User and Oven objects are selected)

3.2 Pattern-Based Trace Query Analysis

3.2.1 Overview

Even filtered versions of MSCs can only facilitate the analysis of a relatively small UML statechart diagram. For a complicated model where direct MSC inspection cannot achieve effective analysis, we perform pattern-based trace queries. To perform pattern-based trace queries, system properties are specified using the pattern system first defined in [8], and then the patterns are used as queries against simulation traces to draw a "True" or "False" conclusion. In the case of a "False" result, a marked section of a simulation trace is presented, highlighting the places where the error occurs. The marked section of the simulation trace is provided to facilitate the user in tracing the error in the statechart diagram.

Pattern-based trace query has been implemented in a prototype tool called Simulation Query Tool (SQT). SQT adopts a pattern system [8] to standardize the specification of system properties so that a system property can be specified by selecting a few parameters provided by the graphic interface. Although SQT cannot process an arbitrary system property due to the limitation of the pattern system itself [8], it does allow users to perform analysis on system properties without expertise in formal notations. Because we are applying the pattern system in the context of *simulation* analysis, it is the case that a property being viewed as true (or false) simply means that the property is true (or false) with respect to the simulation traces that drive the analysis. Thus, as noted before, our pattern-based simulation analysis is not as strong as formal verification (over a full state-space). A concrete example of this limitation shows up on one of the experiments discussed in Section 4.2.4. Still, simulation is widely recognized as a valuable analysis method. We can now discuss specifics of the pattern system and then present how query analysis is performed.

3.2.2 Property Patterns

Dwyer, et al. first proposed a pattern specification system for expressing properties for finite-state verification [8]. We will refer to this work as the "original pattern design." The motivation for this research was to tackle a primary obstacle for the transition of model checking technique from research to practice: practitioners being unfamiliar with specification processes, notations, and strategies. A set of property specification patterns was collected based on the experience base of expert specifiers. These patterns should then be able to assist practitioners in mapping descriptions of system behaviors into a formalism of choice or necessity (such as temporal logic). A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system. A pattern consists of a name, a statement of the pattern's intent (i.e., the

structure of the behavior described), mappings into common specification formalisms, examples of known use, and relationships to other patterns. Based on surveys, Dwyer, et al. concluded that most property specifications for finite-state systems are an instance of one of the proposed patterns.

Each pattern uses a scope to specify the portion of the model execution over which a specification should be verified. A scope is defined by specifying the start and/or ending state/event as delimiters for a pattern. There are five kinds of scope: 1) Global (the specification should hold over the entire execution); 2) Before (the specification should hold in the portion of execution before the given state/event delimiter); 3) After (the specification should hold in the portion of execution after the given state/event delimiter); 4) Between (the specification should hold in segments of the execution delimited by the specified state/event delimiters); and 5) After/Until (like the between scope except that a segment continues even if the right delimiter state/event never occurs).

In the original pattern design the patterns are organized in a hierarchy as shown in Fig. 3.5, based on the patterns' semantics [8].

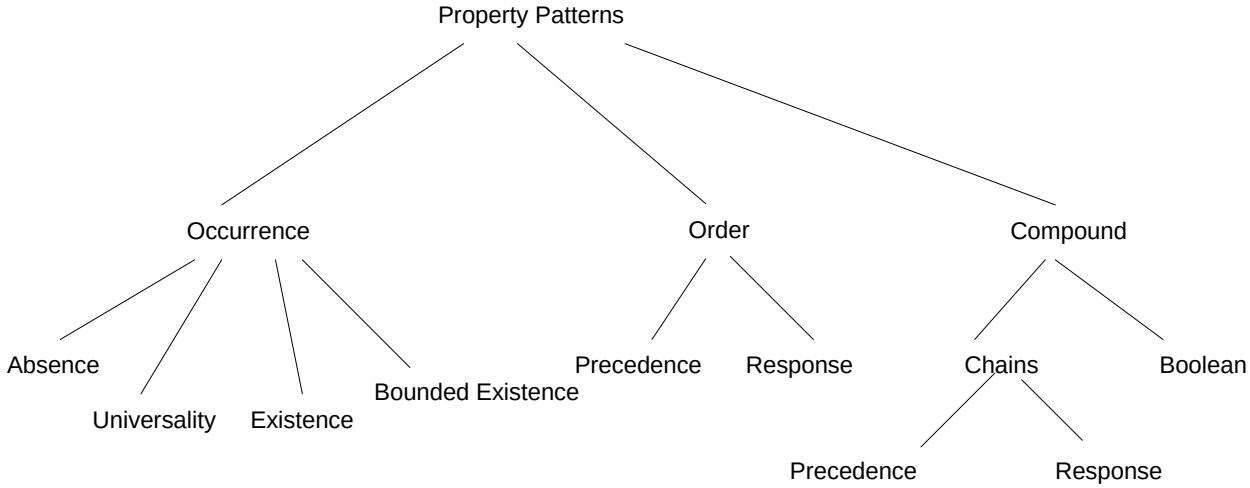


Fig. 3.5 A pattern hierarchy from [8]

In this paper, we refer the patterns under category Occurrence or Order as *simple patterns*, and patterns under category Compound as *composite patterns*. There are six simple patterns: *Absence* (a given state/event does not occur within a scope), *Universality* (a given state/event occurs throughout a scope), *Existence* (a given state/event must occur within a scope), *Bounded existence* (a given state/event must occur k (k >= 1) times within a scope), *Precedence* (a state/event P must always be preceded by a state/event Q within a scope), *Response* (a state/event P must always be followed by a state/event Q within a scope). Category *Compound* has one sub-category *Chains* and one composite pattern *Boolean*, as shown in Fig 3.5. Under the sub-category *Chains*, we have two composite patterns: *(Chained) Precedence* and *(Chained) Response. (Chained) Precedence* means a sequence of states/events P_1... P_n must always be

preceded by a sequence of states/events Q_1... Q_m. This pattern is a generalization of the simple Precedence pattern. *(Chained) Response* means a sequence of states/events P_1... P_n must always be followed by a sequence of states/events Q_1... Q_m. This pattern is a generalization of the simple Response pattern. *Boolean* pattern means a Boolean combination of multiple patterns in Fig 3.5 (may recursively include a Boolean pattern itself).

We implemented all five pattern scopes and the following five pattern types presented in the original pattern design: *Absence*, *Universality*, *Existence*, *Precedence*, and *Response*. To simplify the specification of *Response* under certain instances, we also developed a new pattern – *Constrained Response*. The *Constrained Response* pattern is designed to be a variant of the *Response* pattern, with the restriction that some user-specified "restricted event" must not occur between the pair of events/states (stimulus and response) that comprise the response pattern. This ability to specify restricted events is not supported by the original pattern system. For example, consider a power supply system which supplies power for a light. The power supply system has four events: *power_on* (turn on the power supply), *light_on* (turn on the light), *light_off* (turn off the light), and *power_off* (turn off the power). Under normal operation it is expected that a *power_on* event must be followed by a *power_off* event – this is a typical response pattern. But, to be more precise, we do not want the *light_off* event to occur before the *power_off* event. This would be specified by using the *light_off* event as a restricted event in a *Constrained Response* pattern.

For the simple patterns, only *Bounded existence* is not implemented, because it is a quantitative generalization of *Existence*. (*Bounded existence* means a given state/event must occur k (k >= 1) times within a scope). Query on a composite pattern can be substituted by a series of simple patterns. Therefore, the patterns we have implemented can be used to check all qualitative specifications supported by the original pattern design.

3.2.3 Performing Analysis

We now present our approach to support pattern-based trace query using the pattern system previously discussed. This approach has been implemented in a prototype tool called Simulation Query Tool (SQT), which includes a graphic interface that allows a user to construct queries regarding system properties. The queries are expressed in terms of patterns.

The basic architecture for SQT is depicted in Fig. 3.6. The main components of the architecture are the query engine, which parses the simulation and processes the query, and the graphical user interface, which aids users in the task of composing property specifications. SQT requires two types of files, the simulation-trace files and a model file. A model file is required to allow the tool to provide users with appropriate options for constructing property specifications. Model files are automatically generated from

UML statecharts by the UML-CPN conversion tool. The use of model files in the approach will be elaborated at the end of this section. Once a pattern has been instantiated into a particular query, the queries will then be checked over simulation traces. The simulation trace consists of a series of records, and each record includes: an object, source states, target states, and the associated events. Details for the format of trace file can be found in [15]. The result for a query can be "True" or "False."

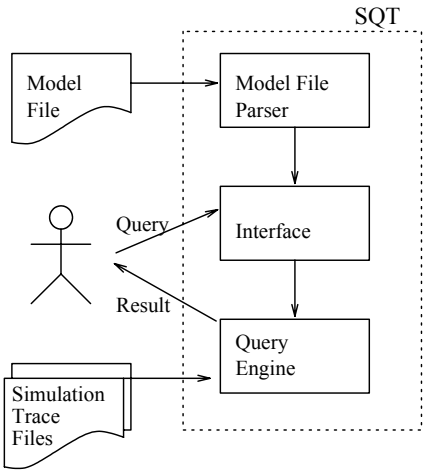


Fig. 3.6 Simulation query tool (SQT) architecture

A key step in performing query analysis is the construction of queries to be checked against simulation traces. In SQT, designers use a custom interface to specify systems requirements by selecting the pattern type, the event parameters and the scope. We refer to the event parameters as "events of interest." Given an informal requirement, besides identifying the events of interest, the designer must determine the combination of pattern type and scope that is most suitable for verifying that requirement. We call the combination of 1) a pattern type, 2) events of interest, and 3) a scope, an *SQT query expression*. As an example, consider the following informal requirement: *When a lock is requested, it must eventually be granted*. Again, we emphasize that since we are operating within the context of simulation analysis, this property, like all subsequent properties, must be interpreted to mean: "Within each simulation trace, when a lock is requested, it must eventually be granted."

From the informal requirement, the user can first identify that the events of interest are the requesting and granting of a lock, which we denote as events *Request* and *Grant* respectively. The requirement describes a cause-effect relationship between a pair of events. An occurrence of the first, *Request*, must be followed by an occurrence of the second, *Grant*. Thus, the appropriate specification pattern is the *Response* pattern. Finally, since the requirement does not state specifically when this relationship should hold, it is reasonable to assume a *Global* scope for this case. Therefore, the SQT query-expression is as follows:

Grant *Responds* to Request *globally*.

Consider another example requirement for the microwave oven system referred to in Section 3.1: *When the microwave-oven door is opened, if the power-tube is energized it must de-energize eventually*.

As before, the user must first identify the events of interest. The events are *DoorOpened* and *Deenergize*. Like the first example, this requirement can be checked using a *Response* pattern. However, since we only expect the *Deenergize* event to follow *DoorOpened* event while the *power-tube* is energized, the scope cannot be *Global*. Instead, the *After/Until* scope must be used. An SQT expression for this property might be as follows:

Deenergize *Responds* to DoorOpened *After* Energize *Until* Deenergize

Here the scope, *After Energize Until Deenergize*, marks the segments of the execution where the power-tube is energized. However, since *Deenergize* is one of the events of interest, *Deenergize* cannot be used for any of the delimiters. Therefore, we need an alternative event that also signals the end of the energize condition. The oven *PowerOff* event serves this purpose. The new SQT expression is as follows:

Deenergize Responds to *DoorOpened* After *Energize* Until *PowerOff*

Another, possibly more natural, way to specify the scope for this property is by using state information. For example, it would seem natural to interpret the query as follows: "Deenergize Responds to DoorOpened while PowerTube is energized." However, since "while" is not currently a supported scope type, we again use the After/Until scope to define the trace segment of interest. Since we now want to parameterize the After/Until scope with states rather than events (as we did previously), we need to point out that states in a trace sequence are "global states." Each global state is composed of local states, one for each object included in the model. For example, for the microwave system a state can be expressed in the following format: ((User, S_1)(Oven, S_2)(LightTube, S_3)(PowerTube, S_4)) where each S_i would denote a local state associated with the specific object. Therefore, we can express our example query using the following SQT query:

Deenergize *Responds* to DoorOpened *After* ((User, X)(Oven, X)(LightTube, X)(PowerTube, Energized)) *Until* ((User, X)(Oven, X)(LightTube, X)(PowerTube, Deenergized)).

Here, the symbol "X" is a reserved variable that matches any state; so the User, Oven, and LightTube can individually be in any local state (they are not required to be in some common state), but the PowerTube must be in the specified state. In this particular example, it suffices to detect that the PowerTube object is not in the Energized state by simply checking if the object is in the Deenergized state. But this is because the object only has two states. In a more complex situation, the state query would also be complicated.

From the previous examples, we can observe the general fact that it is not simple to generate appropriate and effective property patterns – one must have a good understanding of both property pattern

semantics and the application-specific role of events and states. In particular, the specification of scopes is very important since scopes are used to define segments of interest within a simulation trace; the query will only be evaluated within the trace-segments defined by the scope. We will see this issue again in the case studies of Section 4. In those examples, we use scopes that are specified only by events, rather than states, to avoid dealing with global (composite) states.

A query is specified using the graphical interface of SQT. The primary usability guideline is to design the graphic interface so that the components (namely a pattern type, events of interest, and a scope) that compose a query are parameterized. The interface provides a set of drop-down lists, so that the components can be specified by selecting a set of options from drop-down lists. This is similar in concept to the technique used in the Bandera tool [4]. The interface provides a user with options for each component of the query. The options available in the graphic interface can be classified into two categories: *static options* and *dynamic options*. *Static options* are general options independent of the model to be analyzed, while *dynamic options* are customized with model-specific information, contained in the model file. Fig. 3.7 illustrates how to use the interface to specify the above property as a query. The figure also shows that the result for this query is "True."

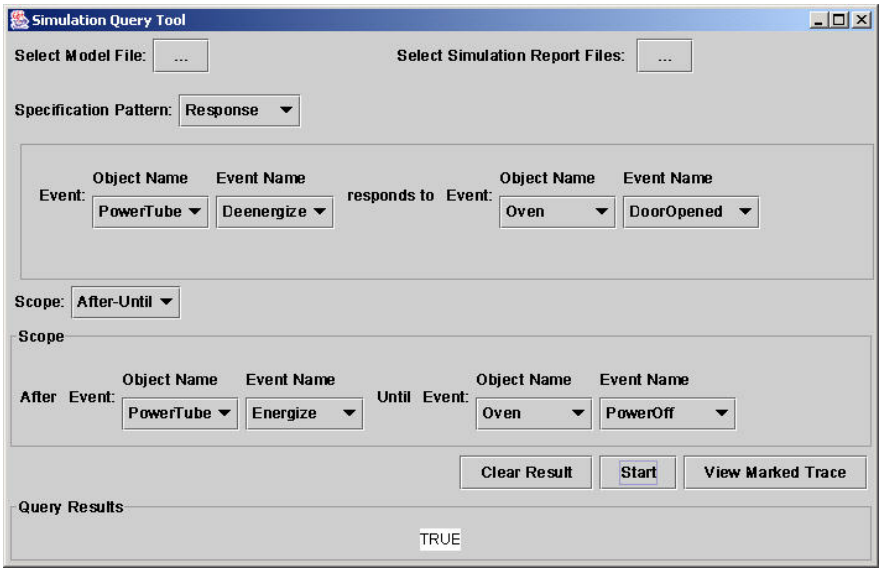


Fig. 3.7 Screenshot of a response query with after/until scope

A model file contains basic information that defines statechart models, such as objects, states and events. A key role for model files is providing parameters for patterns. More specifically, a model file is used to identify parameters that are used to populate the graphical interface for property specification. After the user selects a model file, this file is parsed. As a result, the objects that compose the system model are identified. In addition, the *associated events* of each object are also identified. The identified objects and their associated events are then used to populate the drop-down lists for objects and events,

respectively, on the graphical interface. The *associated events* for an object are those events that can trigger transitions associated with the object. For instance, consider the simulation trace shown in Fig. 3.8. The trace defines that a transition associated with the *User* object has fired, resulting in a state change of the *User* object and the generation of a new event called *ButtonPushed*. Also, after the *Oven* object receives this new event, the event triggers a transition associated with the *Oven* object. As a result of this transition firing, the *Oven* object moves from the state *IdleWithDoorClosed* to the state *InitialCookingPeriod*, and two new events *TurnOn* and *Energize* are generated. The event *ButtonPushed* is an associated event of the *Oven* object. Note that *ButtonPushed* is not an associated event of the *User* object, although *ButtonPushed* is generated by the *User.*

```
User:
    Idle -> Pushing
    | ButtonPushed
Oven:
    IdleWithDoorClosed -> InitialCookingPeriod
    ButtonPushed | TurnOn Energize
```

Fig. 3.8 An example simulation trace

3.2.4 Some Query Processing Details

A simulation trace is a sequence of transitions that have fired during the simulation run. The simulation trace can be viewed as a sequence of states (system states), events, and actions (event generation actions). We call states, events, and actions, *elements*. The trace sequence consists of elements and can be denoted as in the following example:

s0, e1, a1, s1, e2, s2, e3, a2, a3, s3, e4, s4, e5, s5, e6, s6, e7, s7,…… (1)

In this section, we discuss the three main issues involved in query processing: 1) segment extraction from the simulation trace based on the scope specified in the query. A segment is a list of elements that appear within the specified scope. 2) The query algorithm, which looks for some specific element or element pattern on the segments in order to determine a "True" or "False" answer to the query. 3) The analysis of multiple simulation traces.

In order to extract segments for a specific scope, we must decide first whether or not either end of a segment should be open. Given scope types such as *Global*, *Before*, *After*, *Between*, and *After/until*, what are the criteria for determining the closure-ness of scopes? As the original pattern design noted, for an event-based formalism, open-left/open-right scopes are suitable [8]. For a state-based formalism, the original pattern-design uses close-left/open-right because "they are relatively easy to encode in

specifications and they have been the most commonly encountered in the real property specifications we studied." [8].

Two questions are important in order to extract segments. The first question is whether the approach is state-based or event-based. Our approach seeks to verify properties over simulation traces, where a simulation trace contains information about states and events, and a property is specified based on the pattern system, which deals with state/event sequences. So, our analysis approach is event-based as well as state-based. The second question is whether absolute criteria exist for determining the closure-ness of scopes. Will it be theoretically wrong if someone chooses one closure-ness over the other when specifying properties? Or, is it just a matter of convenience. Or, does it depend on the specific properties for a specific system? It seems that the type of closure that is needed varies from properties to properties. The original pattern design states that "Nearly all of the non-global specifications we collected were left-closed right-open, but three were left-open right-closed." [8]. It does not seem very plausible to consider that all the needed properties would have the same closure-ness. To deal with this, we give options to the users and let them decide. Our tool is designed to provide a default closure-ness, open-left/open-right, as well as an interface for the user to change the closure-ness among the following options: open-left/open-right, open-left/close-right, close-left/open-right, and close-left/close-right.

We now describe informally the basic scheme for determining segments for each type of scope. To simplify matters, we only consider event queries and we abstract traces to be sequences of events. The following two generic traces of events are used as sample simulation traces in describing the scheme for the different scopes:

1. P → A→B → C → Q → A→P→Q
2. P → A→B → C → Q → A → P →B→C

Global Scope: For this scope, with either trace 1 or 2, the entire list of events in the simulation trace will be returned as a single segment.

Before Scope: For this scope, the events up to, but not including, the first occurrence of the specified scope delimiter event will be returned as a segment. For example, with trace 1, if the specified scope is "before A," the returned segment would contain only the event *P*. If the delimiter is *P*, then a null segment will be returned.

After Scope: For this scope, all events after the first occurrence of the specified scope delimiter will be returned. With trace 1, if the specified scope is "*After A,*" the returned segment would contain the sequence *B→C→Q→A→P→Q*.

Between Scope: For this scope, the simulation trace will be divided into segments delimited by the specified delimiter events. If the specified scope is "*between A and C,*" then using trace 1, the returned segment would contain the event *B*. Using trace 2, two segments would be returned, the first containing the event *B* and the second containing the sequence $P \rightarrow B$. In general, if neither delimiter occurs in a simulation trace, a null segment will be returned.

After/Until Scope: For this scope, the simulation trace will be divided into segments delimited by the specified delimiter events, except that the segment need not be delimited to the right – the second delimiter event does not have to exist. If the specified scope is *"After A Until C,*" then using trace 1, two segments would be returned. The first containing the event *B* and the second containing the sequence $P \rightarrow Q$. In general, if the first delimiter event never occurs in a simulation trace, a null segment will be returned.

To determine the result for a query, the execution segments are sequentially examined until an error has been found or all the segments have been examined. In general, the query result for each segment must be "True" for the overall result to be "True." In other words, if the query result for any segment is "False," the query result returned by the tool is "False." For each of the six featured pattern types, we now describe informally the process for checking or determining the query result over one segment.

Absence (e.g., *Absence e*): For this type of query, the execution segment is searched for the first occurrence of the specified event *e*. If the event is found, the result of the search on the current segment will be "False;" otherwise "True."

Existence (e.g., *Exist e*): For this type of query, the execution segment is searched for the first occurrence of the specified event *e*. If the event is found, the result of the search on the current segment will be "True;" otherwise "False."

Universality (e.g., *Universality s*): In our work, this type of query is only used for states. To check this type of query, each state element contained in the execution segment is compared to state *s*. If there is at least one state element that is not equal to *s*, the result of the search on the current segment will be "False;" otherwise "True."

Response (e.g., *e2 Responds to e1*): For this type of query, the execution segment is searched for the last occurrence of some event *e1* to check if that event is followed by some other event *e2*. If no event *e1* is found, the returned result is "True," but with a warning indicator that no *e1* was found. If event *e1* is found, then a search is initiated from that last occurrence of *e1* for the event *e2*. If *e2* is found, then the result of the search for that occurrence of *e1* will be "True;" otherwise the result for that *e1* occurrence is "False." For example, the response to query "F Responds to E" is true for the segment "E -> E -> F" but false for the segment "E -> F -> E -> G."

Constrained Response (e.g., *e2 Responds to e1 with constraint e3*): For this type of query, the execution segment is searched for the occurrence of some event *e1* followed by some other event *e2*. The search first looks for the event *e1*. If it finds the event *e1*, it starts to search for the event *e2* from the last occurrence of *e1*. If it finds *e2 and* there is no *e3* between *e1* and *e2*, then the result of the search for that occurrence of *e1* will be "True;" otherwise, that search result is "False." Other details are handled as in the *Response* case.

Precedence (e.g., *e0 Precedes e1*): For this type of query, the execution segment is searched for the first occurrence of some event *e1* to check if that event is preceded by some other event *e0*. If the event *e1* is not found, the search on the current segment will also be "True," but with a warning indicator that no *e1* was found. If event *e1* is found and an event *e0* is found before the first occurrence of *e1*, the search result on the current segment will be "True." If *e1* is found and there is no instance of the event *e0* before the first occurrence of *e1*, the search result on the current segment will be "False." For example, the response to query "F Precedes E" is true for the segment "F -> E -> E" but false for the segment "G -> E -> F -> E."

The complexity of the algorithm for extracting segments from a trace is linear to the size of the trace. The complexity of the algorithm for checking a property against a segment is also linear to the size of the segment.

The segmenting algorithm and checking algorithm described above can be extended to support the analysis of multiple simulation traces. Analysis over multiple simulation traces has the advantage of reinforcing the credibility of the results from the analysis. The more traces are analyzed; the closer is the result to that of the analysis of the reachability graph. Assume that a property specified by a given query needs to be checked across multiple simulation traces. First, the segmenting algorithm is applied to each individual simulation trace so that a set of segments is extracted from the simulation traces. Then the checking algorithm is applied to the set of segments to determine if the property specified by the query holds for all the segments. Consider again trace 1 and 2 above. Assume that the scope is specified as "*between P and C.*" After applying the segmenting algorithm, we obtain three segments. The first, from trace 1, contains the sequence $A \rightarrow B$. The second and third, from sample 2, contain the sequences, $A \rightarrow B$ and *B*, respectively. The checking algorithm can then be applied to the three segments for checking the specified property.

4. Case Studies

In this section, we illustrate our UML statechart diagram analysis framework via two case study examples. In both case studies, a set of properties are checked to study system behaviors. The property

checking can be done two-way – to check if the query result for a desired property is true, or to check if the query result for an undesired property is false. Also, in each case study, one property is highlighted to expose some details on the analysis operations. The main purpose of these case examples is to illustrate the simulation-based analysis capabilities discussed earlier in the paper. Clearly, further case studies using more practical and complex systems are needed in order to make specific claims about the overall effectiveness of these capabilities.

In the first case example – a simple gas station system – we demonstrate how to perform direct MSC inspection, and how to conduct pattern-based trace query analysis on simulation traces. The second case example is more complicated in the sense that it includes composite states; so we start with the pattern-based trace query approach. In that example, a system property needs to be specified by a boolean pattern (a composite pattern). We show how to check this property by considering two simple properties specified by basic patterns. At the end of each case study, we summarize all properties analyzed for the case.

4.1 The Gas Station System[3]

The system consists of a customer and a pump that processes customer requests. The customer must prepay for the gas. After making a payment, the customer is allowed to select the gas grade and then press the nozzle 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 greater 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. Statecharts for the *Customer* object and the *Pump* object are shown in Fig. 4.1. A few comments on the statechart model are warranted: 1) As we will discuss shortly, these models contain some error to be revealed by our simulation analysis; 2) Initially, the *Customer* object is in the state *Idle*, and the *Pump* object is also in its state *Idle*; 3) The unlabeled transition from state *FillingStarted* to state *FillingDone* is a completion (triggerless) transition; 4) Although multiple unlabeled transitions emanating from one state are not typically used in statecharts, the two transitions from state *FillingDone* are purposefully left unlabeled to explicitly create a nondeterministic choice of how to deal with the customer's change since this example does not keep track of the prepaid amount; and 5) Although it may be the case that the example can be expressed via a simpler model, we use this model in order to exercise our simulation approach.

[3] This example is adapted from a specification and partial statechart solution provided by N. Medvidović (Department of Computer Science, Univ. of Southern Calif.), with permission.

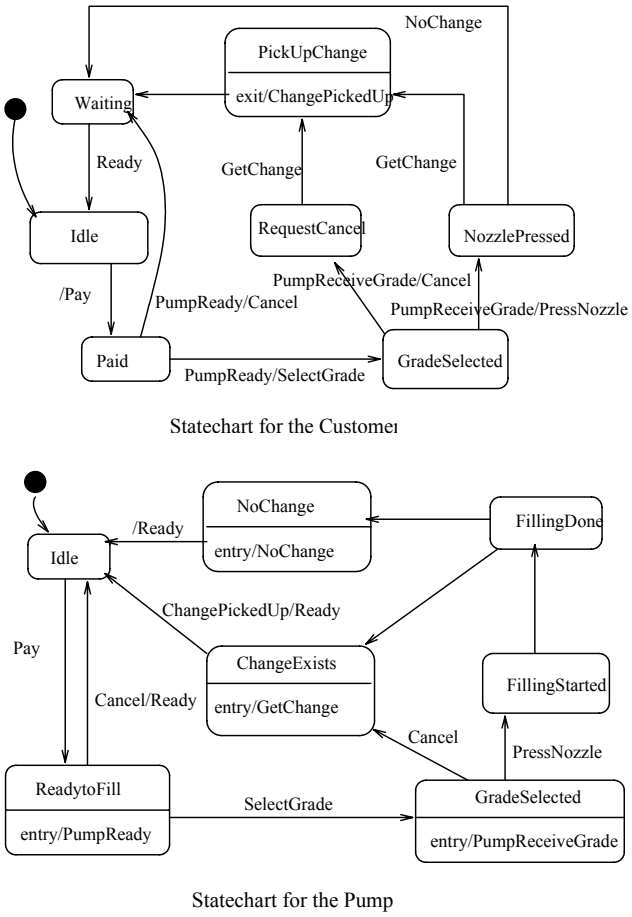


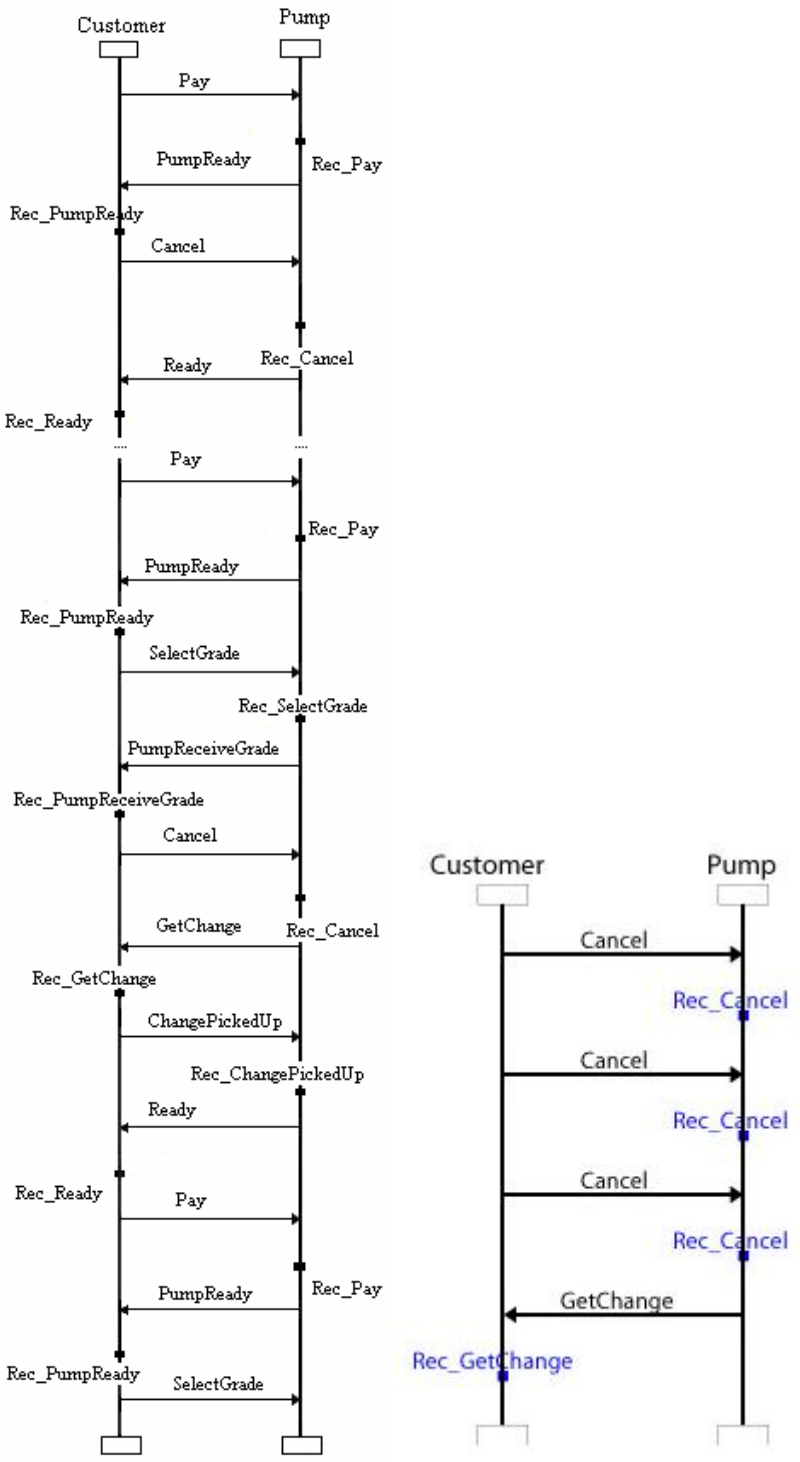
Fig. 4.1. Statecharts for the gas station example

4.1.1 Direct MSC Inspection

For illustration, consider the following property: *If the customer cancels the request for purchasing gas, the customer's prepaid amount should be returned.*

This property can be checked by directly inspecting the MSCs generated from simulation runs. Fig. 4.2 (a) shows a MSC generated by a simulation run. 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 remove unwanted information from MSCs. We use the interface provided by our UML-CPN conversion tool to select the two events of interest. Now, the MSC generated from a simulation run is shown in Fig. 4.2 (b). We can clearly identify that neither of the first two 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. We will discuss how to fix the error in Section 4.1.2.

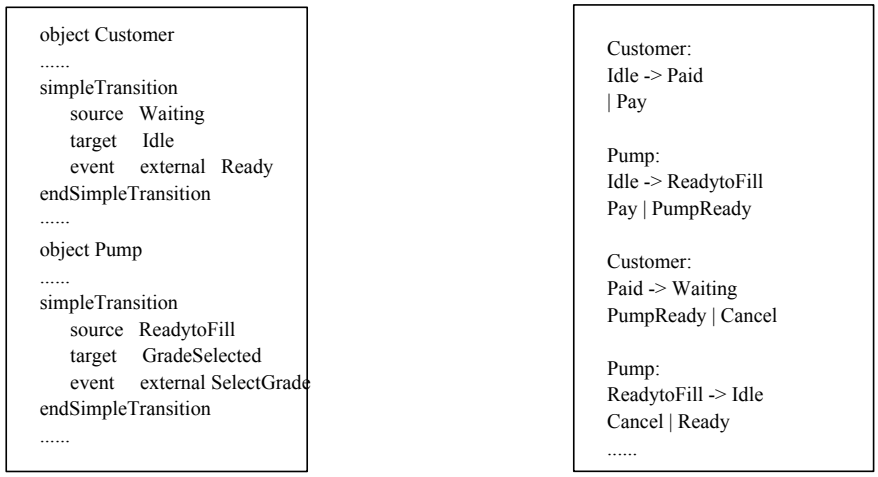


(a) (b)

Fig. 4.2 Message sequence charts for gas station example

4.1.2 Pattern-based Trace Query

In applying our UML-CPN conversion tool to the Customer and Pump objects, we obtain a *model file* and a CPN *model*. Figure 4.3(a) shows parts of the model file, which simply identifies the key elements that make up the source objects. The CPN model is imported into the Design/CPN tool, which generates simulation-trace files. Part of a simulation trace is shown in Fig. 4.3(b), which gives a trace associated with the firing of four transitions. First, a triggerless transition associated with the Customer fires resulting in the event *Pay* being generated. The Pump receives the *Pay* event, and then the event triggers a transition, resulting in the generation of the event *PumpReady*. After the Customer receives the event *PumpReady*, (s)he changes his/her decision and cancels the request for filling gas. This is reflected by the firing of the third transition that consumes the event *PumpReady* and generates the event *Cancel*. After receiving the *Cancel* event, the pump enters the state *Idle* and the event *Ready* is generated.



(a) The model file (b) A simulation trace

Fig. 4.3 The model file and simulation trace for gas station example

To prepare for simulation analysis, the model file (partially shown in Fig. 4.3(a)) is used to populate the SQT graphic interface with two objects: Customer and Pump. We can now discuss how the SQT tool can be used to check various system properties. For illustration, consider the following property (a property we have in mind): *Once the customer cancels the request for purchasing gas, the customer's prepaid amount should be returned.*

Here, the events of interests are the events *Cancel* and *GetChange*, which prompts the Customer to pick up the change – in this case, it is a complete refund. The event *Cancel* is generated by the Customer and responded to by the Pump. Thus, the *Cancel* event is associated with the Pump object. On the other hand, the *GetChange* event, generated by the Pump and responded to by the Customer, is associated with

the Customer. According to the property specification, an occurrence of the *Cancel* event must be followed by an occurrence of the *GetChange* event. So, the Response pattern can be used to check the property. Although the specification does not state explicitly when this property should hold, it can be reasoned that the property should not hold arbitrarily within a simulation trace (in which case the *global* scope would be appropriate to use), but that the property is intended to hold for each complete transaction between the Customer and the Pump. We can adopt the *between* scope to define a transaction segment, but we must then identify events that mark the start and end of each transaction. The event *Ready* plays this role. Note that we can use the same event as left and right delimiters to specify a scope. In this case, the roles of the left and right delimiters are played by two consecutive occurrences of the marker event, in this case, the event *Ready*. Therefore, the SQT query-expression for the property is as follows:

Event GetChange responds to event Cancel between event Ready and event Ready.

The result for this property query is "False," which means that the specified property does not hold for certain transactions between the Customer and the Pump based on the analysis of the simulation trace. We remind the reader that the simulation traces are query-independent. Since traces are not terminated based on any scope (scope is a query-dependent parameter), a trace may contain incomplete transactions even though these incomplete transactions are irrelevant to the query. For the example above, the incomplete transactions are ignored by the query processing algorithm when it considers the scope parameter.

In order to help the designer locate errors in a model, the SQT tool allows the designer to view a marked simulation trace in the case of a "False" query analysis. A marked simulation trace is the same as the original trace except that some transitions are highlighted – recall that a simulation trace records information associated with transitions. If a property does not hold within some segment of a trace, the result for the corresponding query is "False." Consequently, within that segment, those transitions that are triggered by events of interest are highlighted with yellow color (but showing up as light shading in a black-and-white printout). Also, the two transitions associated with the delimiter event(s) are highlighted with green color (showing up as darker shading in a black-and-white printout). Fig. 4.4 shows a marked simulation trace corresponding to our example. As can be seen, there are two highlighted transitions – the first is yellow (light shading) and the second one is green (darker shading). The first transition is associated with one of the events of interest, the event *Cancel*. The second transition is associated with the right delimiter – the event *Ready*. Earlier in the trace (but not shown) is another highlighted (green) transition associated with the left delimiter, also *Ready*.

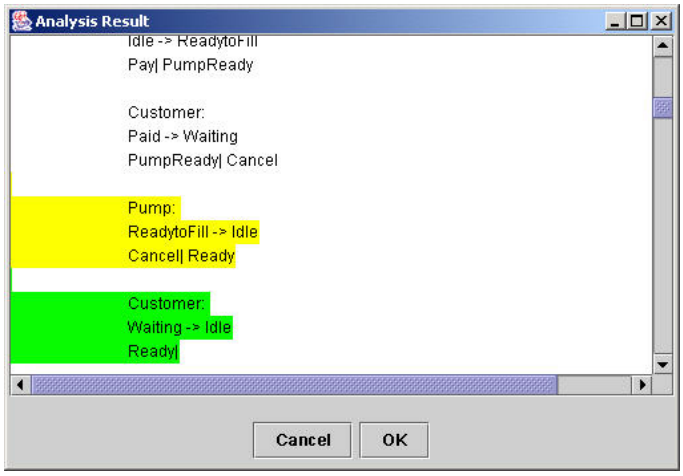


Fig. 4.4 Screenshot of a marked simulation trace for gas station example

From the problematic segment of the trace, we see that no *GetChange* event follows the event *Cancel*; In other words, we only have one "yellow event" between the two marked "green events." By examining the statecharts shown in Fig. 4.2, we can conclude that the statecharts require the following modification. In the statechart for the Customer, the transition originating from the state *Paid* and targeting the state *Waiting* should be replaced by a transition originating from the state *Paid* but targeting the state *RequestCancel* (with the same transition label). Furthermore, in the statechart for the Pump, the transition originating from the state *ReadytoFill* and targeting the state *Idle* should be replaced by a transition originating from the state *ReadytoFill* but targeting the state *ChangeExists*, and the transition label should be modified to remove the *Ready* event. To validate the modified model, we have repeated the experiment. In this case, the result given by the SQT tool for the same query is "True."

Consider what happens if the Pump statechart is modified as indicated above, but the Customer statechart is not modified. In this case, the simulation will terminate, having progressed to a point where the Customer is in the Waiting state (i.e., expecting to receive *Ready*) and only the event *GetChange* (a non-expected event) is available. This *GetChange* event would be found at the end of the simulation trace (and MSC). From this unexpected behavior, the designer has the opportunity to identify the error and then modify the Customer statechart.

4.1.3 Experimental Summary

In our experiment, we collected four simulation traces and we set the simulation tool (Design/CPN) to limit each collected trace to a maximum of 200 steps (i.e., Petri net transition firings). This maximum was chosen arbitrarily, but set to be large enough to allow multiple transactions within each trace; and this was verified by manual inspection of the traces. Since this example models a cyclic behavior of

transactions, execution paths will be infinite. Thus, each of the collected simulation traces was a 200-step trace.

We summarize the experimental results for the gas station case in Table. 4.1. In terms of the pattern-based trace query analysis, for each property tested by an experiment, we provide an informal description of the property, the corresponding SQT query-expression, the query result, and a few comments. In the SQT query-expression, an event is represented in format *(a, b)*, where *b* is the event name and *a* is the associated object. For example, *(Customer, Ready)* is the event *Ready* associated with object *Customer*. Since the gas station diagram models the whole life-cycle of a gas station system and we are only interested in the properties within the scope of a transaction between the Customer and the Pump, we use event *(Customer, Ready)* as the boundary for a transaction.

System Property		Result and Comment
Informal Description	SQT query-expression	
(1) If the customer cancels the request for purchasing gas, the customer's prepaid amount should be returned	Event (Customer, GetChange) *responds* to event (Pump, Cancel) between event (Customer, Ready) and event (Customer, Ready)	False. This is the case discussed previously.
(2) Making payment is a mandatory step during any transaction	(Pump, Pay) *exists* between event (Customer, Ready) and event (Customer, Ready)	True. The property is a desired property.
(3) If the customer pays exact amount, no change should be returned in the transaction after pumping is finished	Event (Customer, GetChange) is *absent* between event (Customer, NoChange) and event (Customer, Ready)	True. The property is a desired property. (See comment (1) for more explanation)
(4) Customer must select grade before lifting the nozzle	Event (Pump, SelectGrade) *precedes* event (Pump, PressNozzle) between event (Customer, Ready) and event (Customer, Ready)	True. The property is a desired property.
(5) Customer can press nozzle to pump before pay	Event (Pump, PressNozzle) *precedes* event (Pump, Pay) between event (Customer, Ready) and event (Customer, Ready)	False. The property is not a desired property.

Comments: (1) Since our model does not track the actual amount of money paid, we use the occurrence of *NoChange* event to represent that the customer used the exact amount and has finished pumping, while the absence of *GetChange* event is used to infer that no change is returned. So, the proposed SQT query-expression can verify that the customer will not receive any change after the customer has paid the exact amount and has finished pumping.

Table. 4.1 Summary of the gas station example experiments

4.2. An Early Warning System

Naturally, to perform full system analysis, it is necessary to first model all objects of the system. But, it is reasonable to expect that partial system modeling and analysis would be valuable during early stages of system design. The example here demonstrates this type of partial system analysis. The early warning system case includes only one object (also named as EWS), although this object has complex

semantics due to its use of composite states. The statechart being considered includes multiple composite states, including a history state and various forms of entry and exit transitions. For this example, we perform a set of pattern-based trace simulation experiments, which are summarized in Section 4.2.4. We start by describing in detail, in Section 4.2.3, one such experiment to help check the correctness of the history state modeling.

4.2.1 Source Statechart

Consider the statechart of EWS in Fig. 4.5, adapted from the early warning system model [11]. Although this statechart is a classical statechart, we interpret it as a UML statechart. As described in [11], the EWS monitors a signal arriving from outside, checks whether its value is in some predefined range, and if not, notifies the operator by an alarm and appropriate messages. The EWS is a general type of system that can be used in a variety of applications for reactive systems.

In order to illustrate the 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. For instance, the event *reset* causes the system to leave both *Comparing* and *Generating_alarm* and enter state *Waiting_for_command.*

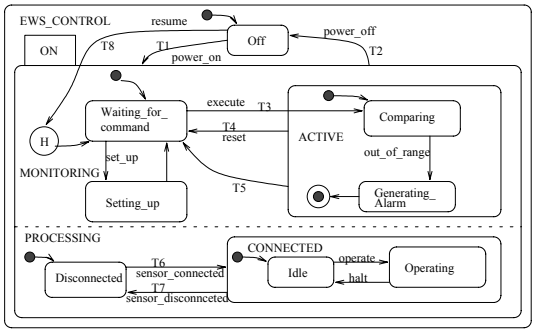


Fig. 4.5 A statechart from the early warning system (EWS) example

4.2.2 Target Model

As mentioned in Section 2, the conversion of composite states is divided into two steps: intermediate model generation and target model generation. The target model is the final resulting CPN model. Fig. 4.6 shows part of the target net model for the composite state EWS_CONTROL. The top portion of the Fig. 4.6 is an event dispatching model, which is generic to all translations, but whose details are beyond the scope of this paper (see [14]). The bottom portion of the figure shows three net structures representing the target models for statechart transitions T1, T8, and T2. According to Fig 4.5, T1 and T8 are entry transitions, and transition T2 is an exit transition.

The target model contains two primary classes of net transitions – those that directly correspond to UML statechart transitions and those that support the modeling of UML statechart 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. 4.6 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. For example, when the critical transition corresponding to *T3* in Fig. 4.5 is fired, the corresponding code segment for *T3* is executed and the following information is recorded into a trace file:

EWS:
Waiting_for_command -> ACTIVE(Comparing)
execute

From the recorded information, we know that in Object EWS, an event *execute* has been processed, so that it triggered some critical transition. As a result, the current state of MONITORING is updated from *Waiting_for_command to ACTIVE.* Since *ACTIVE* is a composite state, the current state in *ACTIVE*, which is *comparing,* is also recorded in the trace file. We also write a pseudo event by the code segment for the purpose of the experimenting. A *pseudo event* is not a real event that is dispatched; it is just an item written to the trace file by code segments to record the firing of a specific critical transition. Transition *T5* in Fig. 4.5 is fired when an alarm is generated and the current state of MONITORING is switched to *Waiting_for_command.* Although no event is generated at this step, we write a pseudo event *leave_active* in the trace file to record the firing of *T5*.

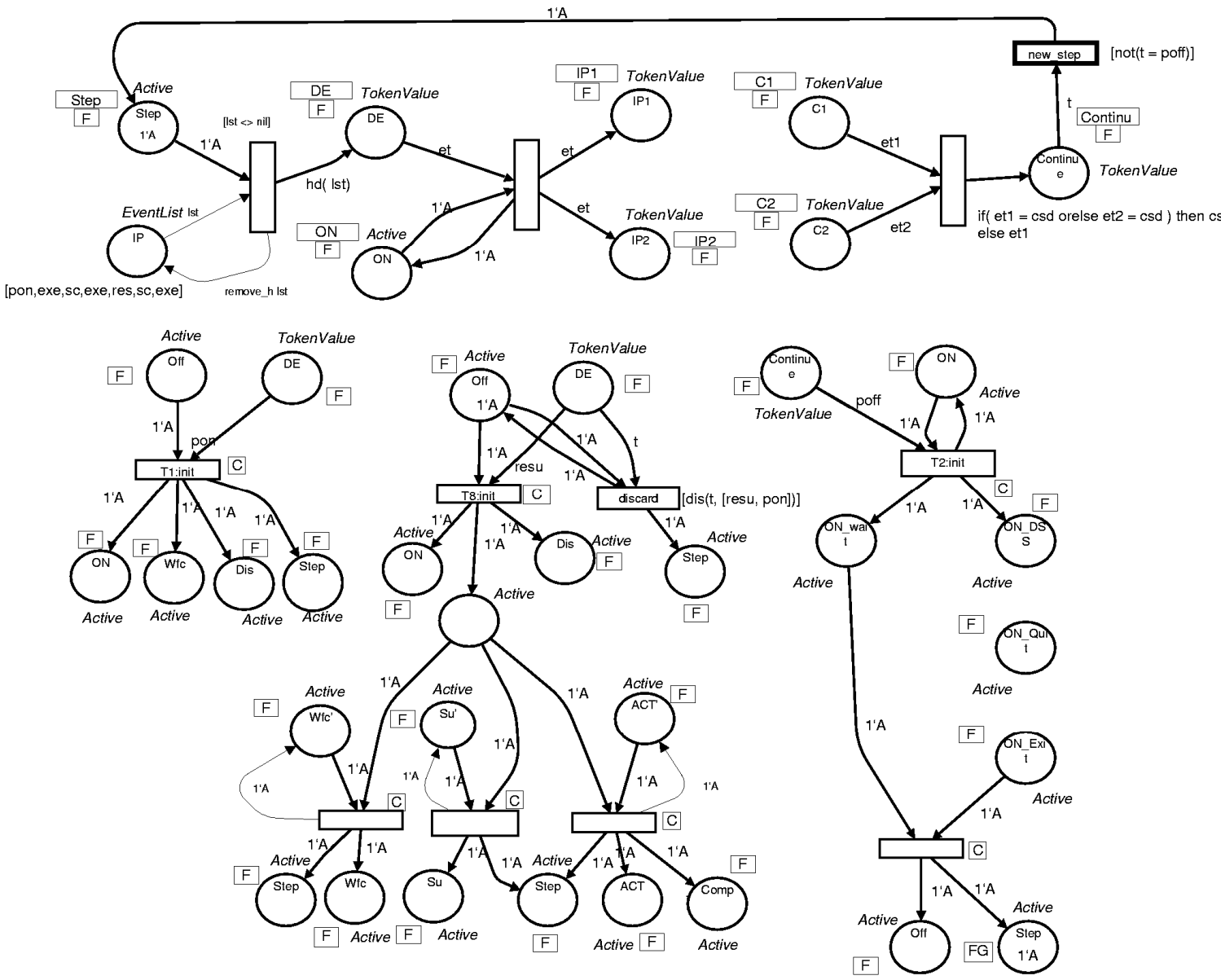


Fig. 4.6 The partial target net model for state *EWS_CONTROL*

4.2.3 Model Simulation and Testing of History State

Since there is only one object in this example, no other objects send or receive events. Thus, to perform our simulation, we first manually generated a list of event tokens and set the generated event token list as the initial token of dispatcher place *IP* in Fig. 4.6. During the simulation, the event tokens are automatically dispatched from *IP*, one after another, to the rest of the target model for processing. EWS does not transfer any event to other objects. According to our conversion technique, only processed event tokens will fire corresponding critical transitions. Therefore, the simulation trace file records only the events actually processed during a simulation run, and those discarded events will not be included in the

simulation trace. Also, the simulation trace will contain the pseudo events *leave_active*, which are generated by the code segments for a critical transition, as mentioned at Section 4.2.2.

For this experiment, we designed the event token list with knowledge of the queries we wanted to evaluate. Thus, the following principle guided our creation of the event token list: The list of events will ensure that events critical to a specific query are actually processed and recorded in the trace file if the model is converted correctly. For example, in order to test the history state, the event list should guarantee that at least one *resume* event token is included in the event token list and it is actually processed to make the history state active.

Since this example involves the conversion of a history state, we demonstrate an experiment to test the behavior of the history state. According to Fig. 4.5, if the history state is modeled correctly, a reset event must be processed between any two processed *execute* events, or an alarm is generated and *T5* is fired, so that *Waiting_for_command* becomes the current state of region MONITORING again. As mentioned before, a pseudo event *leave_active* will be added to the trace file when *T5* is fired. Therefore, the query specification can be written as *Spec1*, given in Fig. 4.7.

Spec1: *(Event reset exists between event execute and event execute) or*
(Event leave_active exists between event execute and event execute)
Spec2: *Event reset exists between event execute and event execute*
Spec3: *Event leave_active exists between event execute and event execute*

Fig. 4.7 A "boolean" pattern based specification

As we can see from Fig. 4.7, *Spec1* is written according to the boolean pattern of two *Exists* pattern. Although we cannot check a boolean pattern directly in our SQT tool, we can write *Spec1* as a combination of two patterns: *Spec1 = (Spec2) or (Spec3)*. *Spec2* and *Spec3* are also shown in Fig 4.7. Both *Spec2* and *Spec3* use *Exists* pattern with the scope *Between* (Between two *execute* events). We can check both *Spec2* and *Spec3* in our SQT tool. For each trace file, we run the SQT tool twice, once to check *Spec2* and once to check *Spec3*. W e found that in all cases either *Spec2* or *Spec3* is true. According to the semantics of "or" operator, we conclude that *Spec1* is true based on all the simulation traces queried.

4.2.4 Experimental Summary

For these experiments, we collected five simulation traces and we set the simulation tool (Design/CPN) to limit each collected trace to a maximum of 300 steps (i.e., Petri net transition firings). It

turned out that each trace was finite and required less than the 300-step limit. The traces were of length 109, 86, 86, 84, and 125 steps.

Besides the history state testing, various other properties were also checked. We summarize the experimental results for the early warning system case in Table 4.2. Like the gas station example, we list an informal description of each property, the corresponding SQT query-expression, and the query result with a few comments. Since there is only one object, we do not explicitly name the object in the query expressions. The analysis results indicate that the conversion is done correctly and that the object's behavior is as desired; although, as we have noted before, such simulation-based analysis does not prove correctness.

System Property		Result and Comment
Informal Description	SQT query-expression	
(1) A history state property	(Event reset *exists* between event execute and event execute) or (Event leave_active *exists* between event execute and event execute)	True. This is the case discussed previously.
(2) To run the system, power must be turned on at least once	Event power_on *exists* globally	True. Existence of event *power_on* is evidence that power has been turned on.
(3) There is no need to resume the power if power is not off	Event resumes is *absent* between event power_on and event power_off	True. The property is a desired property.
(4) The system needs to be setup before start execution	Event setup *precedes* event execute global	False. A somewhat unexpected property, but the statechart does not enforce setup before execution.
(5) If out_of_range event occurs, alarm must be generated	Event leave_active *responds* to event out_of_range between event execute and event execute	True. The property is a desired property. Event leave_active occurs only after alarm has been generated, so it can be used to indicate an alarm is generated
(6) A power loss must be resumed by a resume event, and the system must not restart before the resume event occurs	Event resume *responds* to event power_off globally with constraint power_on	True. The property is a desired property. (See comment (1) for more details)

Comments: (1) The occurrence of a *power_off* event indicates a power loss, and *power_on* event only occurs when the system restarts. Generally, in contrast to supporting a procedure for the sole purpose of power resumption, restarting the system involves more complex operations and is less efficient. Therefore, resuming the power without system restart is a desired property. While this property was true for the particular traces used in our experiment, the statechart for EWS does actually allow for the possibility of using the power_on event to recover the power, which would violate the property. Thus, this example shows the limitation of simulation based analysis.

Table. 4.2 Summary of the early warning system (EWS) example experiments

5. Conclusion and Future Work

We presented a comprehensive UML statechart diagram analysis framework. The core components of the comprehensive UML statechart diagram analysis framework are the UML-CPN conversion process and three analysis operations: direct MSC inspection, pattern-based trace query analysis, and CPN-based model checking. The emphasis of this paper was on simulation-based analysis using direct MSC inspection and pattern-based trace query analysis. These techniques were discussed in detail and illustrated in terms of two case studies. Areas for future work include implementing model checking in the UML statechart diagram analysis framework, performing more case studies to gain further insights about the limitations of the analysis operations, and engaging in controlled user-based usability studies to assess the qualitative value of the user interface and other tool features. Another area for future research is to explore UML diagram consistency checking, especially in terms of automated checking of our automatically generated message sequence charts against existing (manually generated) sequence diagrams that may be available as part of an overall object oriented development methodology. This idea can even extend to checking of our "filtered" MSCs against a designer's set of scenarios. Finally, it would be useful to explore mechanisms that allow for flexible control of simulations that generate trace sequences. For example if certain states or events are viewed as more "critical" to the operation of a system, then it would be valuable to control, or steer, the simulations to exercise such states and events. This extra simulation control may negate the advantage of directly applying a generic model simulator like that associated with many Petri net tools.

Acknowledgement

We acknowledge Robert Parkes for his contributions in the implementation and documentation of the SQT tool, and we thank the reviewers for many insightful suggestions that significantly improved the technical and presentation quality of the described work.

References

[1] L. Baresi and M. Pezzè. "On Formalizing UML with High-Level Petri Nets," In G. Agha and F.De Cindio, editors, *Concurrent Object-Oriented Programming and Petri Nets* (a special volume in Petri Nets series), Vol. 2001 of LNC. Springer Verlag, 2001, pp. 271-300.

[2] 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.

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

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. "Bandera: Extracting Finite-state Models from Java Source Code," *Proc. Int'l Conf. on Software Engineering (ICSE '00)*, IEEE. 2000.

[5] M. L. Crane and J. Dingel. "UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal," *Proc. 8th International Conf. on Model Driven Engineering Languages and Systems*, Oct 2-7, 2005.

[6] Design/CPN. See http://www.daimi.au.dk/designCPN/

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

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. "Patterns in Property Specifications for Finite-State Verification," *Proc. Int'l Conf. on Software Engineering (ICSE '99)*. IEEE. 1999.

[9] H. Gabor, H. and M. Istvan. "Quantitative Analysis of Dependability Critical Systems Based on UML Statechart Models," *Proc. Int'l Symposium on High Assurance Systems Engineering (HASE '00)*, IEEE. 2000.

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

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

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

[13] Z. Hu and S. M. Shatz. "A Transformation Approach for Modeling and Analysis of Complex UML Statecharts: A Case Study," *Proc. of the International Workshop on the Applications of UML/MDA to Software Systems (UMSS05)*, Las Vegas, NV, June 2005.

[14] Z. Hu and S. M. Shatz. "Explicit Modeling of Semantics Associated with Composite States in UML Statecharts," *Journal of Automated Software Engineering*, Vol. 13, No. 4, October 2006, pp. 423-467.

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

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

[17] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Springer Verlag, 2nd corrected printing 1997

[18] 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.

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

[20] 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.

[21] Message Sequence Chart Library, available at www.daimi.au.dk/designCPN/libs/mscharts/.

[22] T. Murata. "Petri Nets: Properties Analysis and Applications," *Proc. IEEE*, 77(4), pp. 541-580, April 1989.

[23] ObjectGEODE. www.csverilog.com

[24] OMG. UML Semantics 1.5, available at www.uml.org.

[25] OMG. UML Semantics 2.0, available at www.uml.org.

[26] 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*, Vol. 1723 in LNCS. Springer, 1999.

[27] 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*, Vol. 1939 in LNCS. Springer Verlag, 2000.

[28] R. G. Pettit IV and H. Gomaa. "Modeling State-Dependent Objects using Colored Petri Nets," *Proc. of Coloured Petri Nets 2001: Modeling of Objects, Components and Agents Workshop*, Aarhus, Denmark, August 2001.

[29] R. G. Pettit IV. "Improving the Reliability of Concurrent Object-Oriented Software Designs," *Workshop on Object-Oriented Real-time and Dependable Systems (WORDS03)*, Capri, Italy, October 2003.

[30] Rational Rose Realtime Documentation. Version 2003.06, Rational Software Corporation

[31] 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.

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

[33] H. Xu and S. M. Shatz. "A Framework for Model-Based Design of Agent-Oriented Software," *IEEE Transactions on Software Engineering*, Vol. 29, No. 1, January 2003, pp. 15-30.