# Specification and Analysis of Real-Time Systems in Statecharts

Sung Deok Cha and Hyoung Seok Hong
Department of Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
373-1, Kusong-dong, Yusong-gu, Taejon, Korea
{cha,hshong}@salmosa.kaist.ac.kr

## Abstract

*Increased use of software in controlling safety-critical systems produced an urgent need to specify and analyze behavior of these systems systematically and rigorously. Statecharts formalism, a popular extension of conventional finite state machines, has been successfully used for specifying requirements of many reactive systems including the TCAS II, an aircraft collision avoidance system.*

*However, little has been published on specific guidelines on how one can best specify and analyze requirements in Statecharts. In this paper, we present a framework for specifying requirements of real-time systems in Statecharts and analyzing them for completeness, consistency, and safety. We use the requirements taken from an emergency shutdown system for a Korean nuclear power plant, called Wolsung SDS2, as an example.*

## 1 Introduction

Software controlling safety-critical real-time systems has already become a part of our society. There are many examples of such systems that impact daily lives of general public. For example, commercial fly-by-wire jetliners (e.g., Airbus A320 and Boeing 777) are currently in service. Many countries operate nuclear power plants, and some, including Korea, are developing software-based emergency shutdown systems. Should emergency situations such as reactor overheating occur, software is required to initiate emergency shutdown procedures while maintaining the plant in a safe state. Some medical devices depend on software to control the amount of radiation to deliver for cancer therapy and to monitor various vital status of patients in an intensive care unit.

When software is used as a control agent in such systems, safety becomes a paramount concern. In the worst case, computer malfunctions, especially unsafe software control outputs, could result in serious and unacceptable consequences such as death, injury, or environmental damage. Software development process is known to be a costly and often error-prone activity, and software quality assurance becomes an essential concern. Of the several phases involved in typical software development, requirements specification and analysis phase is empirically known to play the most crucial role in determining the overall and final software quality.

There are several attributes desired of high quality requirements specification including understandability, maintainability, analyzability, scalability, unambiguity, etc. Many approaches have been suggested on developing high-quality requirements specification. Examples include development of formal specification languages, visual formalisms such as Statecharts[1] or Modecharts[6], and completeness and consistency criteria[5]. These approaches are closely related and tend to be complementary to each other. Formal specification languages enable automated analysis on the requirements but might be difficult to understand. On the other hand, requirements specification in visual languages, while easily understood, might lack formal and unambiguous semantics, thereby making the task of detecting flaws, especially subtle ones, difficult.

Statecharts formalism is a popular language to specify behavior of reactive systems. RSML, an extended version of Statecharts, has been successfully used to specify the behavior of the TCAS II whose logic is very complex[7]. However, there have been few research efforts on methodologies on how software requirements should be organized and specified in Statecharts. Note that a specification language in itself does not provide any methodologies or guidelines. For example, in Statecharts a state can communicate with any of the rest states using events, that is, an event is assumed to be broadcasted to all the states. For another example, the scope of a variable is assumed to be global in that it can be used and defined in the guard and action of any transitions. We have found that undisciplined and unrestricted uses of these features may result in software requirements of poor quality and erroneous.

In this paper, we propose a framework for devel-

0-8186-7570-5/96 $5.00 © 1996 IEEE

oping and analyzing real-time safety-critical systems in Statecharts. Usefulness of our approach is demonstrated using an real-world industrial system, called Wolsung SDS2, a software-based emergency shutdown system whose functionality is similar to that of the Darlington plant in Canada[10].

Our paper is organized as follows: Section 2 briefly reviews Statecharts formalism. Section 3 proposes a framework for specifying and analyzing real-time systems using Statecharts. Additionally, a set of criteria to statically analyze Statecharts is proposed. In Section 4, si afety analysis method for Statecharts, based on forward and backward simulation, is presented. Finally, Section 5 concludes the paper.

## 2 Statecharts

Statecharts, proposed by D. Harel for specifying complex reactive systems[1], consists of $<S, T, E, V>$ where $S$ is a set of *states*, $T$ is a set of *transitions*, $E$ is a set of *events*, and $V$ is a set of *variables*.

States are either *BASIC*, *OR*, or *AND* states: *BASIC* states have no substates. *OR* states have substates that are related to each other by an *exclusive-or* relation. Being in an *OR* state implies being in only one of its substates. *AND* states have substates, called *orthogonal components*, that are related by an *and* relation. Being in an *AND* state implies that being in all of its orthogonal components.

Changes among states are represented by a transition whose format is

*event* [*condition*] / *action*

An *event* is an instantaneous occurrence of a stimulus (trigger), a *condition* is a predicate which should be satisfied for a transition to occur and an *action* may generate other events or perform computations.

In summary, we can say[1]:

Statecharts = finite state machine + depth
+ orthogonality + broadcast

The concept of *depth* is achieved by *OR* states, and *orthogonality* is achieved by *AND* states. *Broadcast* are used for communications between states. The communication is achieved by an action of a transition. That is, when a transition triggered, an action generates an event and this event is assumed to be globally broadcasted.

## 3 A Framework for Real-Time Systems

A real-time system consists of the four basic components: plant, sensors, actuators, and controller (Fig 1). The plant is a physical, mechanical, or electrical process. As far as the controller is concerned, the plant represents the external environment with which the controller interacts. Sensors measure the plant's current state and provide inputs to the controller. Based on the sensor's inputs, the controller performs a set of computations to determine how the current state of the plant should be changed. The controller's decisions are then passed back to the plant by actuators via controlled variables.

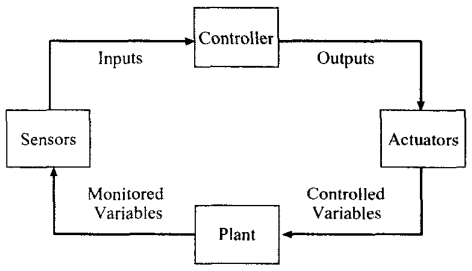

Figure 1: The structure of real-time systems

There is an important distinction to be made between the plant and the controller. The plant is usually "given" in that the developer of the controller may not change characteristics of the plant. In contrast, there is usually more freedom in the development of the controller. In fact, the controller is often implemented by real-time software so that the controller logic can be changed easily if desired. Furthermore, the behavior of the controller cannot be developed unless that of the the plant is first specified.

The controller must regulate the plant's behavior with respect to a set of pre-defined physical laws, objectives, and constraints required of the system. Therefore, requirements of real-time systems should be expressed in terms of not only the controller's functionality but also the plant's behavior. Several methodologies that explicitly take into account these characteristics of real-time systems have been proposed. Examples include the SCR method[11], Heitmeyer and Labaw's method[2]. and TTL/RTTL[9].

However, none of those methodologies supports the use of Statecharts. Several features of Statecharts formalism, when used in unrestricted and ill-disciplined manners, can result in erroneous or in the worst cases, unrealistic requirements specifications. Especially, we have found that the following features of Statecharts can potentially cause ill-disciplined requirements specifications:

- Every event is assumed to be broadcasted to all the states. Therefore, every state can communicate with each other using broadcasted events.

- Every variable is assumed to be global in that it can be used and defined in the guard and action of any transitions.

It should be emphasized that Statecharts in itself do not enforce restrictions or provide guidelines on how various states and transitions should be organized. While one may argue that such complete freedom provides maximum flexibility to the developers, it is often the case that subtle errors caused by the ill-disciplined uses can be quite difficult to detect.

The main purpose of the framework is to properly specify the requirements of the software in real-time systems in a systematic and disciplined manner using Statecharts. The software in real-time systems interacts with the plant, i.e, its environment via input and output variables. The requirements for the software are, therefore, described by specifying the required characteritics of output variables as a response to the observed input variables. In our framework, the software is represented as an *AND* state which has three orthogonal components: *Inputs*, *Functions*, *Outputs* (Fig 2).

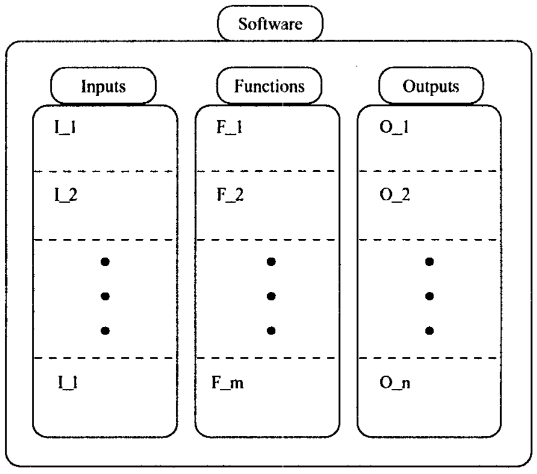

Figure 2: A framework for real-time systems

The first step in our framework is to identify all relevant input and output variables to the software. The behavior of input and output variables is specified by the orthogonal components of *Inputs* and *Outputs*, respectively. Then, the required behavior of the software is identified as a set of functions which are specified by the orthogonal components of *Functions*. We use the prefix "i_", "f_" and "o_" for the orthogonal components states, triggering events, and variables in *Inputs*, *Functions* and *Outputs*, respectively.

We propose several criteria on how to use Statecharts constructs and semantics for specifying the behavior of input, output variables and functions in a disciplined way. After the behavior of all input, output variables and functions are specified, the next step is to statically analyze Statecharts using the following criteria.

**C.1** The orthogonal components of *Inputs* and *Outputs* should have a default state specified with a conditional connective. In contrast, the orthogonal components of *Functions* should have a deterministic default state.

Default states of the orthogonal components of *Inputs*, and *Outputs* represent the state of the plant at the time of software initialization. No assumptions can be made about the default state of the plant, because the changes of states of the plant may occur independently on the software. Therefore, when the software is initialized, every state in the orthogonal components of *Inputs* and *Outputs* may be a possible default state. In contrast, default states in *Functions*, should be designed to have a pre-defined deterministic value, because they represent the initial configuration of the real-time control software.

**C.2** The format of transitions in *Inputs* must be

$$i\_e\ [true]\ /\ f\_e$$

Transitions in *Inputs* represent a change of values of input variables. The triggering events in *Inputs* are external events which are generated outside the software, i.e., in the plant. Therefore they should not directly trigger other transitions in *Functions* or *Outputs*. Note that it is enforced by the naming conventions because only the events whose prefix is "i_" can trigger transitions in *Inputs*. The guard of transitions should be *true* because Transitions in *Inputs* is triggered unconditionally when an external event occurs. The action should generate one and only one event which triggers transitions in *Functions*. In our framework, the behavior of input variables should be specified without any considerations of how they will be used in *Functions*. That is, the logic of the software should be confined in *Functions* for the modularity of the entire specifications.

**C.3** The format of transitions in *Functions* must be

$$[i\_e \mid f\_e \mid timeout \mid \lambda]\ [condition]\ /\ [\lambda \mid f\_e \mid o\_e]$$

where $\lambda$ is a null event.

A transition in *Functions* represents certain computation taking place by the software. Therefore, the triggering event must have been generated externally by *Inputs* ($i\_e$), internally by other parts of the *Functions* ($f\_e$), or implicitly by the *timeout* event and $\lambda$ event. When the computation is completed, it can optionally trigger transitions in *Functions* or *Outputs*. The latter models the generation of external output to the plant. However, generated events may not trigger a transitions in *Inputs*.

**C.4** The format of transitions in *Outputs* must be

$$o\_e\ [true]\ /\ \lambda$$

Transitions in *Outputs* represent a change of values of output variables. The triggering events for transitions in *Outputs* should be generated only by transitions in *Functions*. Because the actuators are passive in nature, transitions in *Outputs* should be triggered unconditionally (i.e., the gaurd is *true*) and should not perform any computation or generate any events on its own.

The criteria **C.2, C.3, C.4** can be summarized as follows: only three types of communications, i.e., broadcasting of events, between states are allowed:

- from *Inputs* to *Functions*,
- from *Functions* to *Functions*, and
- from *Functions* to *Outputs*.

All other types of communication are to be prohibited.

**C.5** Each input variable should be used at least once in the software and each output variable should be defined at least once by the software.

If an input variable is not used at all in *Functions*, then it is an indication of a missing functionality in the software. Similarly, there is a missing functionality in the software if an output variable is not defined at all in *Functions*.

**C.6** Every state must have a transition defined for every possible event, i.e., conditions of outgoing transitions should be complete.

**C.7** Every state should have one and only one corresponding transition for every possible event. That is, conditions of outgoing transitions should be deterministic to be considered consistent.

Jaffe et al. discussed completeness and consistency equivalent to the criteria **C.8** and **C.9**[5] in terms of conventional finite state machines. Recently, Heimdahl and Leveson proposed similar concepts for RSML[3] and developed an analysis method for these properties.

## 3.1 A Case Study: Wolsung SDS2

To demonstrate our framework and analysis methods, we use a real-world application, Shutdown System Number 2 (SDS2) of Wolsung Nuclear Power Plant. as an example. The SDS2 monitors the state of the nuclear reactor states such as pressure and power and generates a trip signal if the nuclear reactor becomes unsafe, i.e., the pressure is too high or too low. There are several trip parameters in SDS2 and we describe only the *PDL delayed trip parameter*[1] as follows:

- If *i_Pressure* $\leq$ 950 kPa (delayed trip setpoint) and *i_Power* $\geq$ 80%FP, then continue normal operation without opening *o_PDLTrip* for three seconds.
- After three seconds has expired, the value of *o_PDLTrip* becomes *Open*, if *i_Power* $\geq$ 80%FP. Otherwise, the value of *o_PDLTrip* becomes *Close*.
- Once *o_PDLTrip* has been opened, keep it *Open* for 1 seconds.
- Then the value of *o_PDLTrip* becomes *Close* once *i_Pressure* $>$ 950 kPa or *i_Power* is $<$ 80%FP.

In this example, there are two input variables: *i_Pressure*, *i_Power* and one output variable: *o_PDLTrip* and one function: *f_PDLDelayedTrip*. Fig 3 shows a Statechart for the *PDL delayed trip parameter* part of the SDS2.

Transitions in *Inputs* are triggered by the external events and generate events which are broadcasted into only *Functions*. The default state of i_PressureState is determined by a conditional connective which contains predicates over the corresponding input variable, *i_Pressure*. The *Outputs* consists of *o_PDLTrip* which represents whether the trip parameter is open or close. Note that the events *o_open*, *o_close* should be generated only by transitions in *Functions*. The behavior of *f_PDLDelayedTrip* is as follows: the default state is $S_1$ which represent that the controller is idle and in waiting state. Once the state $S_2$ is entered, the software waits for three seconds. After three seconds the software checks the current values of *i_Pressure* and *i_Power* and determines whether or not to generate a trip signal.

## 4 Safety Analysis of Statecharts

Safety analysis techniques often employ backward analysis approach where a hazardous system state

[1]For simplicity, we have slightly simplified the actual requirement.

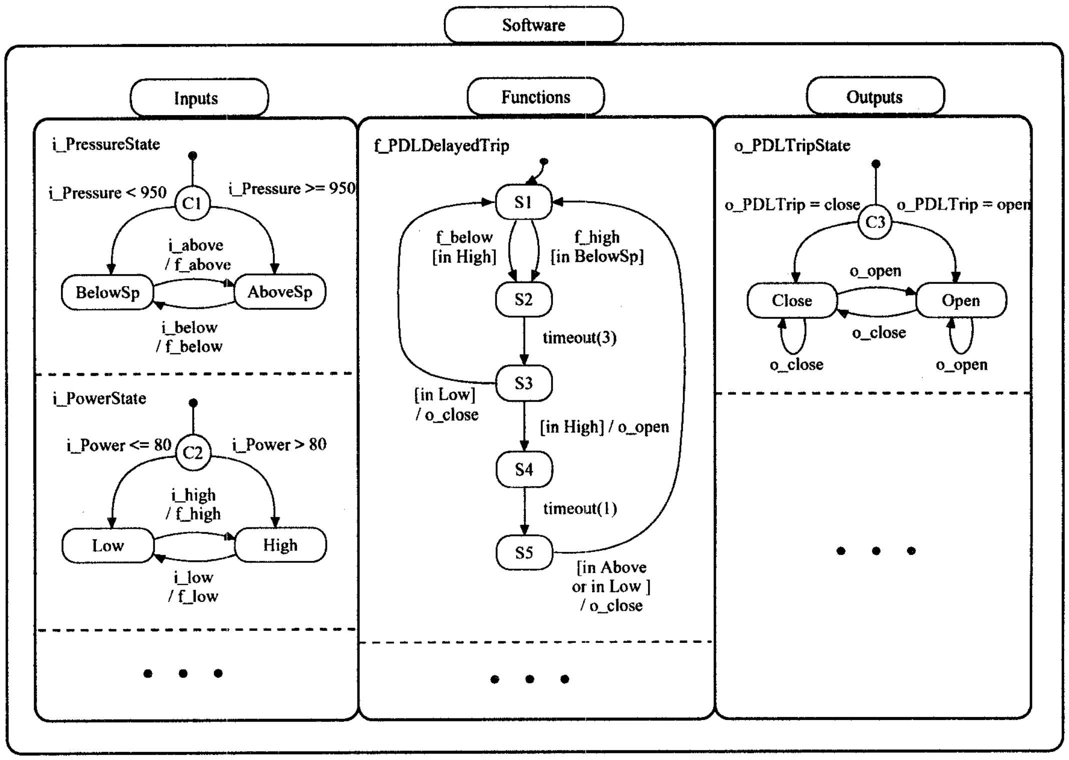

Figure 3: A Statechart for Wolsung SDS2

is assumed to have occurred and where the credible causes and their relationships (when applicable) are explored. When performing fault tree analysis, backward analysis is usually performed manually without any assistance of automated tools. Therefore, the effectiveness of the FTA method depends heavily on the capability of system safety analysts.

An alternative is a model-based backward analysis approach using Petri nets[8] or other state-based models. Backward step-by-step simulation can provide analysts with deeper understanding of if and how the hazardous system states might occur. Once event sequences leading to hazards are identified, they can be eliminated or risks minimized by introducing additional safety requirements.

## 4.1 Forward Simulation

The semantics of Statecharts [4, 12] are based on the *synchrony hypothesis* which assumes that the system is infinitely faster than the environment. Hence, the response to an external events is always generated in the same time when the event is introduced. The semantic description of Statecharts is based on the notion of *micro-step*. A micro-step is initiated when an external event arrives at the system, causing a cascade of subsequent internal events. A micro-step is completed when no more internal events are generated or there are no more transitions triggered by the events that were previously generated.

Unfortunately, when applying forward simulation to verify safety property of complex systems, *state explosion problem* is encountered. While our framework does not completely solve the state explosion problem, various restrictions on the transition formats may reduce the complexity of forward simulation considerably. In order to further simplify the forward simulation process, we distinguish the two types of events:

- *Spontaneous* events refer to the events "i_e" in *Inputs* and *timeout*, $\lambda$ in *Functions*. The occurrence of these events does not depend on any other component in Statecharts.
- *Non-Spontaneous* events refer to the events "f_e" in *Functions* and "o_e" in *Outputs* These events may be generated only as a consequence of actions associated with events triggered by spontaneous events.

For example, in Fig. 4, *i_below* is a spontaneous event and *f_below* is a non-spontaneous event.

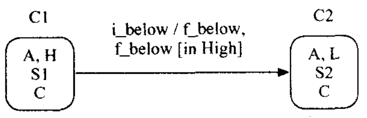

Figure 4: An example of spontaneous and non-spontaneous events

In summary, every event in *Inputs* is spontaneous and every event in *Outputs* is non-spontaneous. Events in *Functions* can be either spontaneous or not. Transitions in *Functions* whose triggering event is *timeout* or $\lambda$ represent a spontaneous function. Transitions whose triggering event is *f_e* represent a non-spontaneous computation initiated by the sensor or by other function of the controller.

We note that any changes in the Statecharts configuration can be initiated only by a spontaneous event and forward simulation can be conducted by identifying all spontaneous events in the current configuration. Note that this forward simulation procedure is different from that of pure Statecharts where there is no notion of spontaneous or non-spontaneous events.

An example of forward simulation is shown in Fig. 5. Assume that the current configuration is $c_0$ = $(A,\ L,\ S_1,\ C)$[2] There are two spontaneous events which trigger outgoing transitions from $c_0$: *i_below* and *i_high*. Secondly, we generate the next configurations based on the micro-step semantics for each event.

## 4.2 Backward Simulation

In contrast to forward simulation that usually starts from a initial system configuration, backward analysis typically starts from the system configuration that is assumed to be hazardous. For example, a configuration $(B, H, *, C)$ means that whenever the system is in the states $B$, $H$, and $C$, it is hazardous regardless of the states of the *f_PDLDelayedTrip*. Note that the states in *Functions* need not necessarily be included in the definition of the hazardous system state because the system could be in the hazardous state regardless of the software's state. In this case, the configuration should be regarded as follows:

$$(B, H, *, C) = \{(B, H, S_i, C) \mid 1 \leq i \leq 5\}$$

Once the hazardous system configuration is defined, backward analysis can proceed by considering

[2]For notational convenience, we denote states in Fig 3, *AboveSp*, *BelowSp*, *Low*, *High*, *Close*, *Open* with *A*, *B*, *L*, *H*, *C*, *O*, respectively.

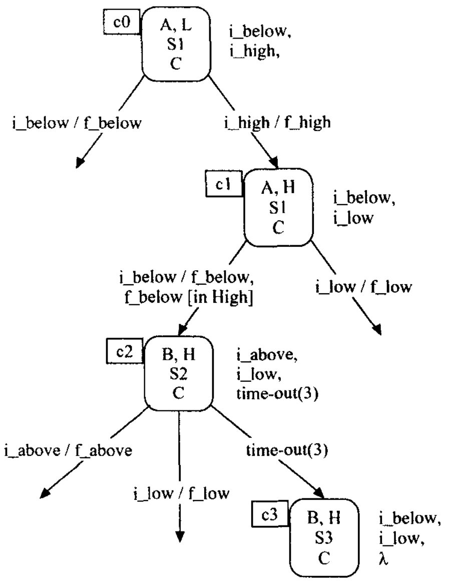

Figure 5: An example of forward simulation

only spontaneous events. As an example, let's assume that the hazardous configuration from which backward analysis begins as $c_0$ = $(B,\ H,\ S_5,\ C)$ which means that "trip signal is not generated when the value of *i_Pressure* $\leq$ 950 kPa, *i_Power* $\geq$ 80 %FP and the software is in the state $S_5$" (Fig. 6). There are three spontaneous events which trigger incoming transitions to $c_0$: *m_below*, *m_high* and *timeout*(1). For each event, the preceding configuration can be generated. For example, taking a step backwards following the event *i_below*, $c_1$ = $(A,\ H,\ S_5,\ C)$ is obtained.

To demonstrate how the $\lambda$ event can be properly handled, let's take a step backward from the configuration $c_2$. Brute-force and reverse application of the micro-step semantics would result the configuration $c_3$ = $(B,\ H,\ S_3,\ C)$. However, this path is infeasible because the transition [*in High*] / *o_open* would result in a configuration $(B,\ H,\ S_4,\ O)$, not $c_2$, due to the action of the transition, *o_open*. Hence, we need not further continue the backward analysis from $(B,\ H,\ S_3,\ C)$. In Fig 6, infeasible paths are shown in dashed line.

## 5 Conclusions

In this paper, we have proposed a framework to specify and analyze the behavior of real-time systems in Statecharts and demonstrated its effectiveness using Wolsung SDS2 shutdown system. It should be

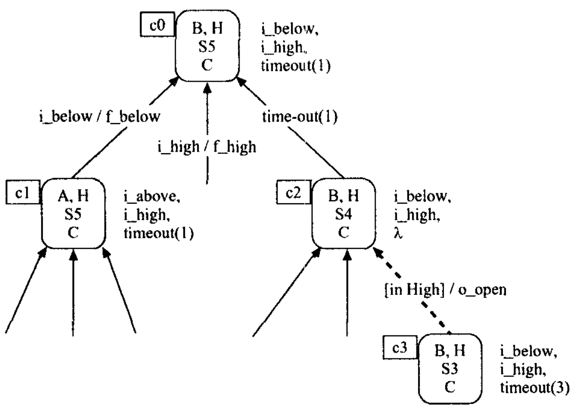

Figure 6: An example of backward simulation

emphasized again that Statecharts formalism does not provide any guidelines or restrictions on how one can properly specify and analyze the behavior of real-time systems. In our framework, we provide several guidelines and enforce several restrictions on how the features of Statecharts are used. It is important to note that such restrictions have been introduced for systematic specifications of software requirements based on the inherent characteristics of real-time systems.

In our framework, software requirements are described by specifying the behavior of output variables in terms of input variables using three orthogonal components: *Inputs*, *Functions*, and *Outputs*. Briefly reviewed, Statecharts are developed and analyzed as follows:

1. Identify all the relevant input and output variables of the plant. The behavior of each input and output variable is specified by the orthogonal components of *Inputs* or *Outputs*, respectively. A set of functions representing the required behavior of the controller is specified next as the orthogonal components of *Functions.*

2. Statecharts are statically analyzed for several criteria such as consistency and completeness criteria. Restrictions enforced on states and transitions reduce the possibility of introducing flaws in the model.

3. Statecharts that have passed the static analysis are then dynamically analyzed. We have demonstrated how forward and backward simulations can be performed in Statecharts. Specific guidelines on formulating the definition of hazardous system states are also suggested.

## References

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

[2] C. Heitmeyer and B. Labaw, "Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier," In *Foundations of Real-Time Computing: Formal Specification and Methods*, Kluwer Academic Publishers, pp. 291-313, 1991.

[3] M.P.E. Heimdahl and N.G. Leveson, "Completeness and Consistency Analysis of State-Based Requirements," in *Proceedings of the 17th International Conference on Software Engineering*, Seattle, pp. 3-14, Apr. 1995.

[4] D. Harel, A. Pnueli, J.P. Schmidt and R. Sherman, "On the Formal Semantics of Statecharts," in *Proceedings of Symposium on Logic in Computer Science*, pp. 54-64, 1987.

[5] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B.E. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Transactions on Software Engineering*, Vol. 17, No. 2, pp. 241-258, Mar. 1991.

[6] F. Jahanian, and A.K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, pp. 933-947, Dec. 1994.

[7] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, pp. 684-707, Sept. 1994.

[8] N.G. Leveson and J.L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. 13, No. 6, pp. 386-397, Mar. 1987.

[9] J.S. Ostroff, "Verification of Safety Critical Systems Using TTL/RTTL," *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, LNCS 600, Springer-Verlag, 1991.

[10] D.L. Parnas, G.J.K. Asmis, and J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety*, Vol. 32, No. 2, pp. 189-198, April-June, 1991.

[11] D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering," CRL Report 237, Faculty of Eng. McMaster University, Hamilton, Ontario, Sept. 1991.

[12] A. Pnueli and M. Shalev, "What is in a Step: On the Semantics of Statecharts," *Theoretical Aspects of Computer Science*, LNCS 298, Springer-Verlag, pp. 244-264, 1991.