Statechart: A Visual Language for Software Requirement Specification

W. Zhang, T. Beaubouef, and H. Ye

***Abstract*—Statechart is a visual language for software requirement specification that has been widely used in recent years. In essence, it extends the conventional language of state-transition diagrams with three elements that accommodates the notions of hierarchy, concurrency, and communication. Additionally, it allows multilevel concurrency, creation of chain-reaction effects, and the use of high- and low-level events. It is compact, expressive, compositional, and modular. This paper presents a tutorial of Statechart and discusses its merits and shortcomings.**

***Index Terms*—Statechart, requirement specification.**

I. Introduction

A software requirement specification is the first detailed documentation of the required behavior of a software system. Errors introduced during the requirements analysis phase are difficult and expensive to correct if they are allowed to be propagated into the design phase. Such errors may cost up to 200 times more to correct than those introduced later in the life cycle [2] and can have major impacts on the system.

A requirement specification is a behavioral specification of the system's activities; it describes the system's modes (states) of operation and events that cause the system to change modes. The specification often includes a set of assertions that must be satisfied. These assertions are invariant properties of the software system, and thus should also be included in the requirement specification.

The requirement specification is particularly critical for complex embedded software. Embedded software is part of a large system and has a primary purpose of providing at least partial control of the system or process in which it is embedded. Most such software is real-time and reactive, i.e., required to interact with and respond to its environment in a timely fashion during execution. It is very difficult to specify and validate the requirements that describe reactive behavior clearly and at the same time both formally and rigorously.

The requirement specification must be unambiguous and translatable into mathematical notation, but it need not itself including arcane mathematical symbols that are unfamiliar to the application experts and software developer. The specification should be simple and clear, and it should contain only the information needed by the developer and analysts. The language for software requirement specification should be easy to use and result in more readable and revisable specification.

One approach is the classical formalism of finite state machines and state transition diagrams. A finite state machine (FSM) is a model of a system with discrete inputs and outputs. The system can be in any one of a finite number of internal states or configurations. The state of the system summarizes the information concerning past input that is needed to determine the behavior of the system on subsequent input. One state, denoted by q0, is the initial state. The system consists of a finite set of states and transitions from state to state that occur on input symbols. For each input symbol there is exactly one transition out of each state or there are more than one transitions out of a state. A directed graph, called a transition diagram is associated with a FSM. The vertices of the graph correspond to the states. If there is a transition from state q0 to state p on input a, then there is an arrow labeled a from state q to state p. Fig. 1(a) shows a state transition diagram with four states and eight transitions. A Mealy machine is also a finite state machine, except it gives an output in response to input. Fig. 1(b) shows a Mealy machine, which takes input 0 or 1 and gives output n.

Harel [3] indicates that people cannot use conventional FSM and transition diagrams in designing complex systems for several reasons:

1. State transition diagrams are *"flat"* without notion of depth, hierarchy, or modularity.
2. State transition diagrams are uneconomical for transitions. An event that causes the very same transition from a large number of states must be attached to each state separately.
3. The number of states grows exponentially in state diagrams.
4. State transition diagrams are inherently sequential and are not well suited for concurrency.

Harel [3, 4] constitutes an attempt to revive the classical formalism of finite state machines and state transition diagrams and make them fitting for use in large and complex applications. Statechart: a visual language for software requirement specification is proposed to overcome these drawbacks of state diagrams while preserving and even enhancing the visual appeal of conventional state diagrams.

Statechart extends the conventional language of state transition diagrams with essentially three elements that accommodate the notion of hierarchy, concurrency, and communication. Statechart transforms the state transition diagram into a highly structured and economical description language. When coupled with the capabilities of computerized graphics, Statechart enables people to view the description at different levels of detail and makes very large process-control requirements specification manageable and comprehensible.

Manuscript received November 28, 2011; revised December 20, 2011..

W. Zhang and T. Beaubouef are with Southeastern Louisiana University, Hammond, LA 70402, USA (e-mail: wzhang(tbeaubouef)@selu.edu).

H. Ye is with The University of Newcastle, Callaghan, NSW 2308, Australia. (e-mail: Huilin.Ye@newcastle.edu.au).

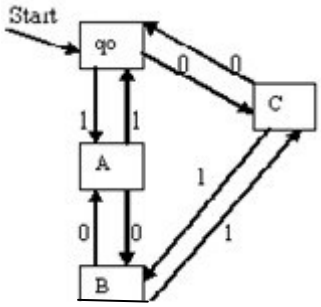


Fig.1(a) FSM

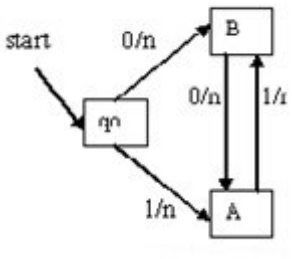


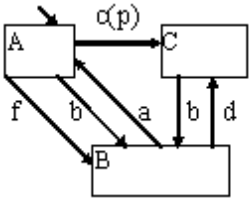
Fig. 1(b) Mealy Machine

The syntax and semantics of Statechart use low level functional formalism. The semantics appears to be novel in its treatment of shared variables, chain-reactions and simultaneous multiple transitions. In recent years, Statecharts have been used as UML (Universal Markup Language) state machine diagrams [1] and state machine notations for control abstraction [19].

This paper presents the features of Statechart. Notations used in this paper are graphics, symbols, and plain English. This paper is organized as follows: Sections 2 and 3 introduce the basic and additional features of Statechart. Section 4 discusses how to represent concurrency in Statechart. Some drawbacks will be discussed and some related approaches will be briefly introduced in Section 5.

II. Basic Features of Statechart

Statechart is a finite state machine augmented with schemes for expressing hierarchy, parallelism, and communication. Rectangles are used to denote states at any level. A simple finite state machine is composed of states connected by transitions. An arrow labeled with an event, and optionally with a parenthesized condition, denotes the transition. A small arrow marks default or start states. In Fig. 2(a), there are three states: A, B, and C. Event c that occurs in state A, transfers the system from state A to state C if and only if (iff) condition p holds at the instant of occurrence. State A is the default state; that means the system enters state A when the state machine is entered unless otherwise specified.



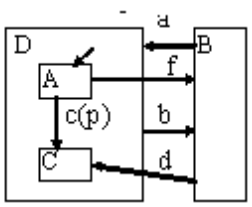


Fig. 2(a) Three States &One Event Fig.2(b) Super State D

A. Composition of a Superstate

In a Statechart, states may be grouped into a *superstate*. The concept of a *superstate* has its origin in *higraph* [4], which combines the notions of Euler circles, Venn diagrams, and hypergraphs. A Statechart may contain states at any level, and encapsulation is used to express the hierarchical relation.

In Fig. 2(a), since event *b* takes the system to state B from either state A or state C, states A and C can be clustered into a new *superstate* D, and the two *b* arrows can be replaced by one as shown in Fig. 2(b). The semantics of D is the exclusive-or (XOR) of states A and C, i.e., being in state D is equivalent to being in either state A or state C, but not both. Superstate D is an abstraction of states A and C. Such groupings reduce the number of transitions needed to be drawn on a Statechart. The *superstate* D and outgoing arrow *b* capture a common property of states A and C, viz., a transition from either of its substates A or C via arrow *b* to state B.

A *superstate* can be entered in two ways. First, the transition to the *superstate* may end at the border of the *superstate* as exemplified by arrow *a* in Fig. 2(b). In that case, the default state A is entered, i.e., it is equivalent to having arrow *a* drawn from state B to state A. Second, the transition may be made to a particular state inside a *superstate*, such as arrow *d* in Fig. 2(b) that leads from state B to state C.

Grouping states into a *superstate* indeed reduces the number of transitions and makes the specification more readable. Furthermore, an economical representation of arrows with common sources, targets, or events is allowed in Statechart as shown in Figures 3(a)-(d). In Fig. 3(d), arrow a splits into two arrows where one leads to state B and the other leads to state C. This is a contradiction to the desired determinism of the system and should not be used.

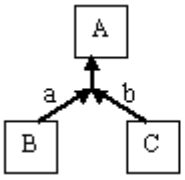


Fig. 3(a) Transition 1

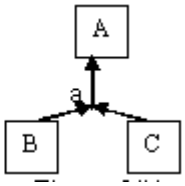


Fig. 3(b) Transition 2

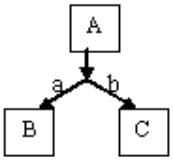


Fig. 3(c) Transition 3

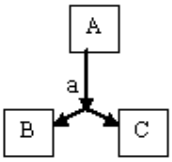


Fig. 3(d) Transition 4

Clearly, more subtle contradictions can occur as a result of the deep character of Statechart and should be carefully avoided. For example, Fig. 4(a) shows an arrow a contradiction from state A. Arrow a leads the transition from state A to state E as well as from superstate B to state F. Fig. 4(a) also contains a default state contradiction upon entering state B via arrow d. There are two default states in superstate B. One is state E and another is state D contained within state C. Arrow c is under-specified, since state C contains no default state.

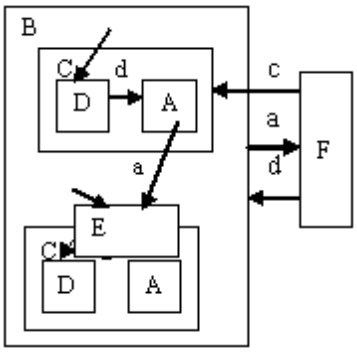


Fig. 4(a) Shows an arrow a contradiction from state A.

In Fig. 4(b), condition (¬ in A) is added to arrow a which is from the border of superstate B to state F. If the current state is A, arrow a leads the transition from state A to state E; otherwise it will lead the transition from state B to state F. In Fig. 4(b), state D is the default state of superstate C. Arrow d will lead the transition from state F to state E, and arrow c will lead the transition from state F to state D all by default arrows.

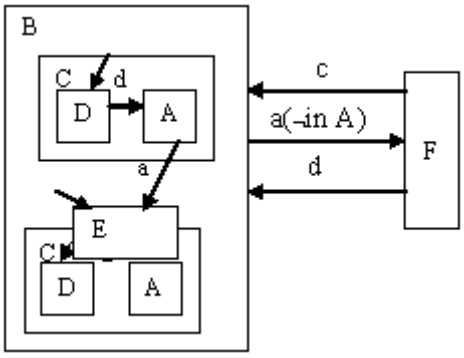


Fig. 4(b) Condition (¬ in A) is added to arrow a

B. History Entry

One of the most interesting and frequent ways of entering a group of states is by the entering history. Statechart has the ability to "remember" a previous visit to a state. Circled H is used to denote the *history entrance*.

The simplest kind of this "enter-by-history" is entering the state most recently visited. In Fig. 5(a), the H-entry with arrow *a* means entering the most recently visited or entering default state D if it is the first time. Fig. 5(b) shows two entrance arrows to superstate A. In this case, the default state D is entered when the entrance is via arrow *a*. If state A is entered via arrow *f*, the state entered is one of three states: state B, state C, or state D. It depends on what state the system happened to be in when it was most recently in state A.

An H-entry generally means that the history is applied only on the level in which it appears. In Fig. 5(a), history entry chooses only between superstates G and F. The system chooses superstate G and enters state B if it was in state A or state B when it most recently left K. The system chooses F and enters state C if it was in state C, state D, or state E on the last visit. The choice of entering state B and state C is by the default arrows in G and F.

If the H-entry wants to override the default all the way down to the lowest level of states, an asterisk is attached to the H-entry. Fig. 5(b) shows that the system will enter the most recently visited state from among states A-E overriding both defaults.

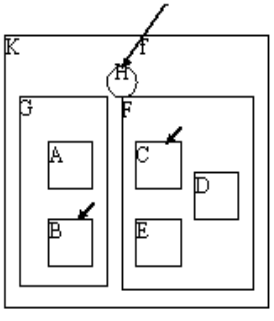


Fig. 5(a) H-entry between G and F

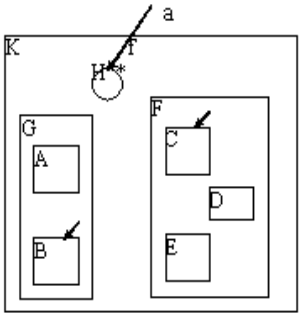


Fig. 5(b) H-entry override default

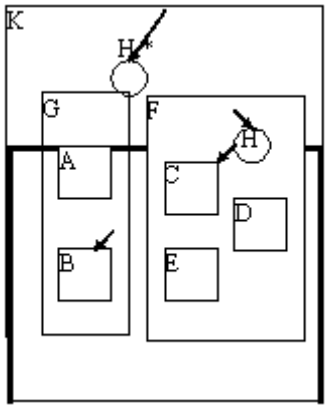


Fig. 5(c) Additional H-entry

To achieve effects in between one-level and all levels, additional H-entries are needed. In Fig. 5(c), the system will enter state B if its last visit to K was to G or that one of state C, state D, or state E was last visited. The default arrow to state C is overridden by history through the H-entry in F.

Sometimes, H-entry should be interpreted with reservation. The history is to be forgotten if a specific event occurs. To deal with this more complex historical criterion, the special actions *clear-history(state)* and *clear-history(state*)* will be used. These actions cause the forgetting of the most recently visited state on the current level, or of all levels of state. In Figure 5(a), *clear-history*(K) will forget the most recently visited states F or G. In Fig. 5(b), *clear-history*(K*) will forget the most recently visited states from among A-E. Once forgotten, H-entry does not apply and defaults are employed.

C. Composition of a Parallel State

One of the most important innovations in Statechart is *parallel state*, which is also referenced as orthogonal state or product state. A parallel state contains two or more parallel components (AND components) separated by dashed lines.

In Fig. 6(a), parallel state H consists of two parallel components, state A and state D. The semantics of H is the product (AND) of states A and D, i.e., being in state H entails being in both state A and state D. Each of the parallel components state A and state D within H is entered whenever the parallel state H is entered. In Fig. 6(a), when parallel state H is entered from the outside via arrow *f*, the substates B of A and substates F of D are entered by the default arrows. When any transition is taken out of the *parallel state* H, all states H, A, and D are exited.

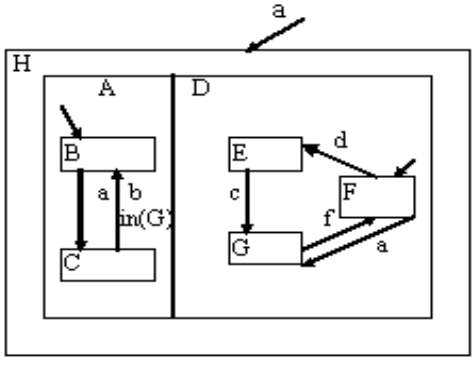


Fig. 6(a) Parallel State H

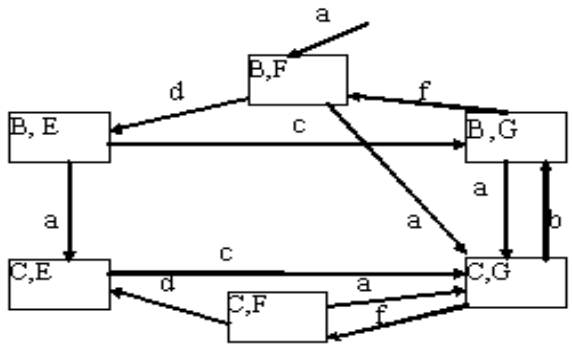


Fig. 6(b) AND-free "flat" Statechart

The *parallel components* state A and D can be *superstates* or be *parallel states* themselves. In Fig. 6(a), *parallel components* state A and state D are *superstates* themselves. *Parallel state* in statechart illustrates a certain kind of synchronization. In Fig. 6(a), if event *a* then occurs, the transition from state B to state C and from state F to G will take place simultaneously.

The use of a *parallel state* greatly reduces the size of the specification. Fig. 6(b) is the conventional AND-free equivalent "flat" version of Fig. 6(a). The usual product of conventional state transition diagram is a disjoint product. Fig. 6(b) contains six states, the product of the two substates in A and three substates in D. Clearly, two components with one thousand states each would result in one million states in the product. This is the root of blow-up in number of states. *Parallel state* in Statechart introduces some dependence between components, i.e., in Fig. 6(a), the special condition "in(G)" attached to arrow *f* causes state A to depend on state D and indeed to "know" something about a substate of D. If the parallel construct is used often, and on many levels, a state explosion problem can be overcome in a reasonable way.

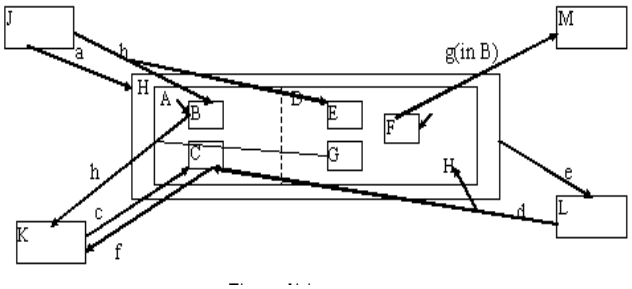


Fig. 6(c) Interface of Parallel State H

A *parallel state* can be entered in four ways. First, the transition to the *parallel state* may end at the border of the parallel state as exemplified by arrow *a* in Fig. 6(c). Fig. 6(c) adds a possible interface description of the *parallel state* H of Fig. 6(a). Internal transitions of Fig. 6(a) have been omitted for simplicity. In this case, as mentioned before, state B and state F are entered by default. Second, the transition may be made to particular states inside the *parallel components*, such as split arrow *b* in Fig. 6(c) that leads transition from state J to state B and state E. Third, the transition may be made to one particular state, such as arrow *c* in Fig. 6(c) that leads to transition from state K to state C by arrow *c* and state F by default. Fourth, the transition may be made to one particular state and an H-entry such as split arrow *d* in Fig. 6(c) that leads transition from state L to state C and the most recently visited state in state D.

A *parallel state* may be exited in three ways. First, the transition exits *parallel state* from the border of *parallel state* as exemplified by arrow *e* in Fig. 6(c). In this case, the *parallel state* H and all *parallel components* state A and state D are exited unconditionally. Second, an "exiting independently" transition exits *parallel state* from an inner state such as arrow *f* in Fig. 6(c) that leaves state H, state A, and state D and enters state K. Third, an "exiting dependently" transition exits *parallel state* from a certain combination of states as exemplified by arrow *h* in Fig. 6(c). In that case, the event *h* that occurred in state B and state G causes transferring from parallel state H to state K. An alternative to the third case is to replace one of the outgoing branches of the merging arrows by a condition as showed in arrow g from state F in Fig. 6(c). In this case, transition exits parallel state H to state M only from state F and state B.

The use of parallel state reduces the state explosion problem in the conventional state machine. The parallel state components can be carried out on any level of states and is therefore more convenient than allowing only single level sets of communicating in FSM. The use of a parallel state enables Statechart to describe independent and concurrent state components and eliminates the need for multiple control activities within a single activity.

D. Conditional Connectives

A condition defines what must be true before the transition can be taken. When transitions out of a particular state into two or more different states are taken based on the same event but different conditions, conditional connectives are used. Circle C is used for abbreviating the conditional connective.

In Fig. 7(a), when event *a* occurs, the transition out of state A enters state B, state C, or state D depending on conditions *P, Q,* or *R.* In Fig. 7(b), a conditional connective, ©, is used to simplify the multiple transition entrances. The transition from the source state A to the connective © is taken at the occurrence of a event. The appropriate destination state, state B, state C, or state D is determined based on the guarding conditions that are defined on the transition from the connective © to the destination states.

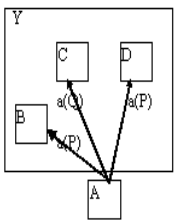


Fig. 7(a) Condition 1

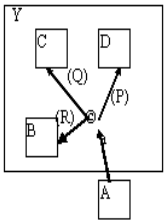


Fig. 7(b) Condition 2

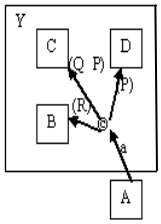


Fig. 7(c) Condition 3

To maintain the completeness and consistency of a specification, the guarding conditions from the connective to the destination states must be mutually exclusive. When event a occurs and condition P is true, Fig. 7(c) shows a contradiction. Since both state C and state D are substates of superstate Y, they cannot be in state C and state D simultaneously.

If all the destination states share some guarding conditions, those conditions may be placed on the transition from source state to connective ©. In Fig. 7(d), all the destination states

share the guarding condition *S*. Condition *S* is placed on the transition from state A to connective © as shown in Fig. 7(e).

Sometimes a state change is not desired. In this case, a transition leads from the conditional connective back to the source state. For example, when event *a* occurs, the transition from state A to the connective © is taken in Fig. 7(b). If none of the guarding conditions *P, Q,* and *R* is true, a transition leads from © back to state A.

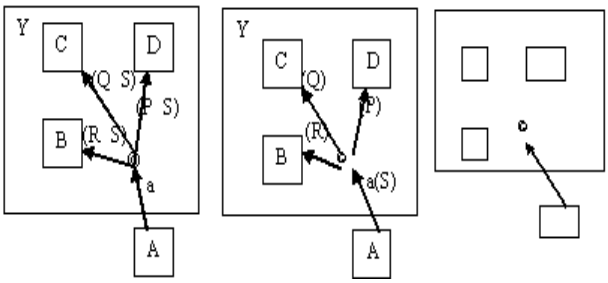


Fig. 7(d) Share Condition S Fig. 7(e) Connective Fig. 7(c) Simple form

If the actual conditions are too complex, details from the chart can be omitted. A simple incomplete form of Fig. 7(f) can be used. The full details of conditions and explicitly specifying the circumstance for changing a state and for remaining in a state can be supplied separately.

III. Additional Features of Statechart

This section contains a number of the more advanced features of Statechart. For most of them, there is neither final formal syntax nor satisfactory formal semantics. These features represent significant potential strengthening of Statechart as a tool for specifying real systems. Basic approaches of these features will be discussed here.

A. Selective Connective

Sometimes a state to be entered is determined as a simple one-to-one by the selection of a generic event. The event is actually the selection of one of the clearly defined options. The user will have to specify the event selection as being the disjunction of the lower-level events and associate each of them with an appropriate state. All those options can be specified as states in a Statechart.

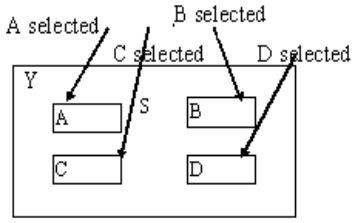


Fig. 8(a) Four Keys

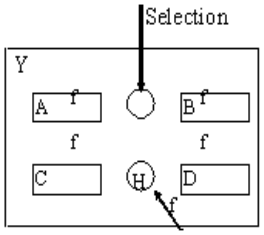


Fig. 8(b) Use Selection

Fig. 8(a) shows the system with four keys marked as A, B, C, and D pertaining to the objects stored, their code names, quantities and physical placements. The system allows the user to select an option by pressing the appropriate key. The system then processes the selected option, possibly repeatedly if no other key is pressed.

Circled S is used to denote the selective connective. In Fig. 8(b), the selection arrow to circled S replaces all four arrows in Fig. 8(a). The unified H-entry simplifies all repeated events. In Fig. 8(b), pressing *f* in any of the substates A, B, C, or D causes exit (but not exit from the encapsulating superstate Y) and immediately entrance to the most recently visited substate. This *f* arrow replaces four *f* arrows, one for each substate in Fig. 8(a). By using *selective connective* and H-entry, Fig. 8(b) simplifies the selections in Fig. 8(a).

B. Timeout and Time Bound

To limit the system's delay in a state and put a time constraint on the state is an important property of a real-time system requirements specification. Statechart uses implicit timers to respond to time restrictions. Formally, this is done using the event expression timeout(event, number). This expression represents that timeout event occurs precisely when the specified number of time units have elapsed from the occurrence of the specified event.

In Fig. 9(a), the system will exit from state A to state B when 120ms have elapsed from the occurrence of event *f*.

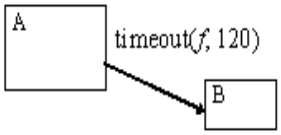


Fig. 9(a) Timeout

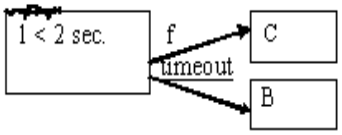


Fig. 9(b) Time Bound

Sometimes the need to limit the system's lingering in a state occurs repeatedly in the specification of real systems. A graphical notation is needed to show this property of a state. Statechart uses a squiggle to indicate that a state comes with a *time bound*.

In Fig. 9(b), squiggle shows that state A comes with a time bound. "1< 2 sec" is an indication of a bound itself. In general, the syntax of the bound specification attached to a squiggle is $\lhd t_1 < \lhd t_2$, providing lower and upper bounds on the time in a state. Either one of $\lhd t_i$ can be omitted. Events do not apply in the state until the lower bound is reached. A generic event *timeout* stands for the event expression *timeout*(*entered A*, *bound*) where state A is the source of the *transition* and the *bound* is its specified bound.

In Fig. 9(b), the lower bound is 1 second. Thus the event *f* cannot cause a transition leaving state A until one second has elapsed from the occurrence of event *f*. The system will exit from state A to state B after two seconds have elapsed from the entering of state A.

C. Unclustering

The conventional notation for hierarchical description has the advantage of keeping the state transition diagram small, yet the parts of interest large. When the system under description is large, Statechart adopts the conventional notation.

In Fig. 10, part of the Statechart can be not within but outside of its natural neighborhood. Fig. 10 shows that Statechart is unclustered into several layers.

When the system under specification is large, it is a necessary option to remain uncluttered. Taking unclustering to the extreme yields a tree structure, thus undermining the basic area-dominated graphical philosophy of Statechart.

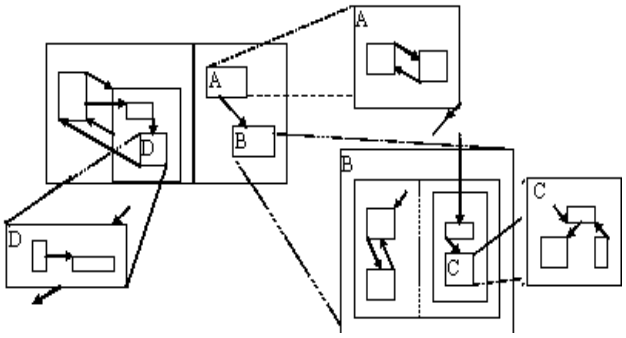


Fig.10. Unclustering

D.State Arrays

In many cases, different states have identical internal structure. Some of the most common ones are best viewed as a single state with a parameter. To choose a notation that economizes by parameterizing the states, state array is used to view the common ones as a single state with an index in Statechart. Individual state array element is referenced by the array name and an index value, i.e., A[3] refers to the third element in state array A.

There are two semantic approaches to state array. One is a "parameterized-and" relationship between the element state of state array. In this case, state array is semantically equivalent to identical parallel states uniquely identified by an index. Each element state of state array is entered or exited when the state array is entered or exited. For example, the requirement specification for each of thirty aircraft in TCAS[13] has identical internal structure. Instead of using thirty different states, Fig. 11(a) shows a *state array* other-aircraft with 30 elements. The aircraft collision avoidance system (CAS) will enter or exit each aircraft simultaneously.

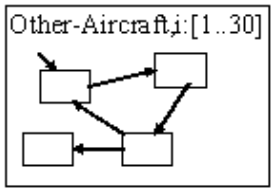


Fig. 11(a) State Array

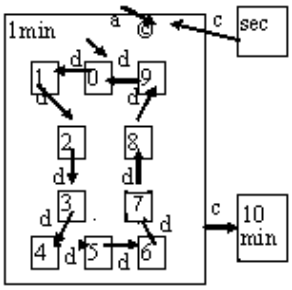


Fig. 11(b) State Update 1minite

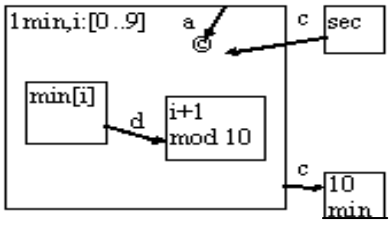


Fig. 11 (c) State Array Notation of Fig. 11(b)

Another approach is a "paramerized-or" relationship between the element states of a *state array*. In that case, the semantics of *state array* is the exclusive-or of element states, i.e., the system can only be in one element state at a time. Consider the state update, 1min [2] as shown in Fig. 11(b). In this state, the condition tests the current time *T*, and the unspecified event *a* denotes *T* crossing a minute borderline. States 1-9 are all updated by event *d*. Fig. 11(c) shows the *state array* notation of Fig. 11(b). Each time event *d* occurs, the state updates to the next element state.

E. Transition Buses [12]

By using *superstate* and *parallel state*, Statechart has the ability to reduce a large number of states to a conceptually manageable number. Transition buses are introduced to reduce clutter in Statechart.

In Fig. 12(a), states A, B, C, and D are fully interconnected, i.e., there is a transition from each state to every other one. In Fig. 12(b), states A, B, C, and D are near fully connected, i.e., there is a transition from state A to every other state and there is a transition from states B, C, and D to every other one except state A. Showing each single transition explicitly in these two charts is confusing and makes the Statechart hard to read.

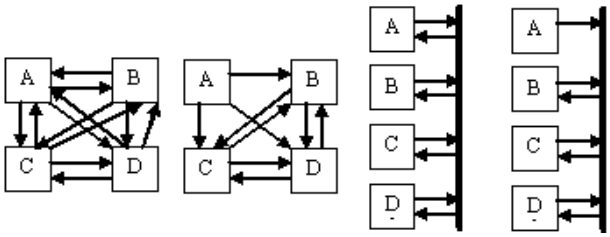


Fig.12(a) Connected Fig.12(b) Near Full Fig.12(c) Bus 1 Fig.12(d) Bus 2

In this case, the transition bus can be constructed to provide the same information. During constructing, a transition must be defined for each source-state and destination-state pair on the transition bus. Source-state is a state with a transition to the bus line and a destination-state is a state with a transition from the bus.

Fig. 12(c) shows a transition bus that is equivalent to the transition of Fig. 12(a). Since all the states are fully interconnected in Fig. 12(a), the states, A-D are both source and destination states. Fig. 12(d) is the *transition bus* for Fig. 12(b). Since there are no transitions from states B, C, and D to state A, there is no transition from the bus to state A in Fig. 12(d).

F. Overlapping State

Superstate and *parallel state* provide exclusive-or (XOR) and parallel (AND) interrelationship between the states in Statechart. Sometimes OR interrelationship between the states may be needed in Statechart.

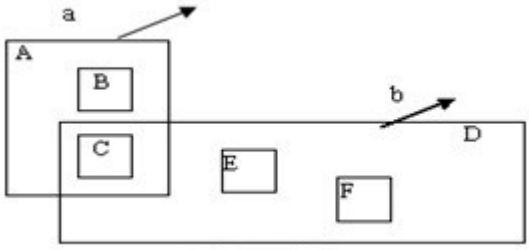


Fig. 13 Overlap States

Fig. 13 shows that substate C could be in state A, or in state D, or in both state A and state D. In this case, state A and state D are actually related by OR not XOR relationship. The OR interrelationship is needed. The reason to have the OR interrelationship might be due to conceptual similarities

between the involved states, or merely the desire to economize when describing joint exits such as the two transitions arrow a and arrow b shown in Fig. 13. In these cases, overlapping states are used to turn XOR's to OR's.

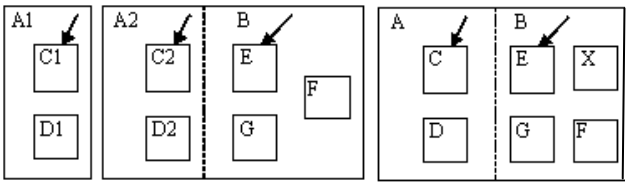


Fig. 14(a) Superstate A Fig. 14(b) State B has X

Consider a *superstate* A with substates C and D that stay alone under some circumstances. State A joins in parallel with state B under other circumstances.

Fig. 14(a) shows one way of describing this case. In Fig. 14(a), state A appears twice. Once for the standalone state A and another for state A parallel with state B. If state A contains many substates and has complex internal transitions, obviously this representation is not a desirable one.

Fig. 14(b) shows another way of describing it. State B contains a special extra new state X that indicates that it is not really a B state at all. This solution is rather artificial, and is difficult to manipulate.

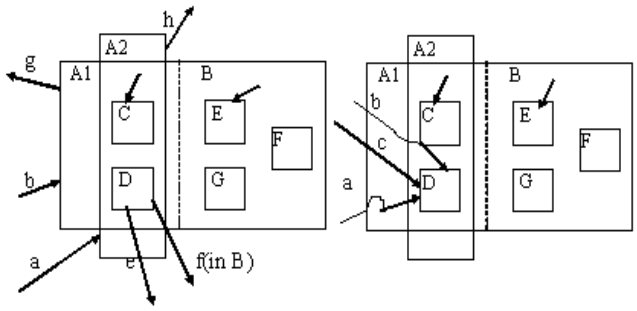


Fig. 15(a) Overlapping Fig.15(b) Add Half Circle

Overlapping states can be constructed to solve this problem as shown in Fig. 15(a). When following the semantics of *superstate* and *parallel state*, all the transitions appearing in Fig. 15(a) are unambiguous. Arrow b is clearly an entrance to the product of state A1 and state B. By default, the system will enter states C and E. Arrow a enters state A2 alone and gets into state C by default. Arrow e leaves state D and state A2 regardless of whether it is matched with a state from state B or not. Arrow f exits from state D and state A2 only when the Statechart is also in state B. Arrow h leaves state A2 alone and arrow g leaves both state A1 and state B.

By using the elementary notation of transition arrows, transitions to substates of overlapping state will be ambiguous. For example, it is not clear what arrow c entering state D is supposed to mean. For example, arrow *c* could enter state D alone or it could enter states D and E. Refine the arrows crossing state borderlines by adding half circles as arrow *a* and arrow *b* shown in Fig. 15(b). The half circle crossing a state borderline means that transition bypasses this state. In Fig. 15(b), arrow *a* bypasses *parallel state* A1 and enters state D only. Arrow *b* bypasses state A2 and will enter states D and E.

Overlapping states can be used economically to describe a variety of synchronization primitives and to reflect many natural situations in complex systems. In other words, too much overlapping states may cause incomprehensibility that outweighs economy of description.

IV. Concurrence

Statechart is generally used to represent the control part of the system. This reaction part is responsible for making the time-dependent decisions that influence the entire behavior of a system. The problem with concurrence stems from the events and conditions that are generated within the Statechart itself. In this section, configuration, chain reaction, and semantics of step transition in Statechart will be discussed.

A. Configuration

Any two states in Statechart can be related in one of three ways: exclusive, parallel, or hierarchy. A set of states s is consistent if and only if (iff) all states in this set hold the properties of XOR and AND interrelationship.

In Figure 16, the sets {A, B, D, F, H, I} and {A, B, D, G, H, I} are both consistent because they both hold the properties of XOR and AND. The set {A, B, C, F, J} is not consistent because state B and state C are both sub-states of *superstate* A. The system cannot be in state B and state C simultaneously.

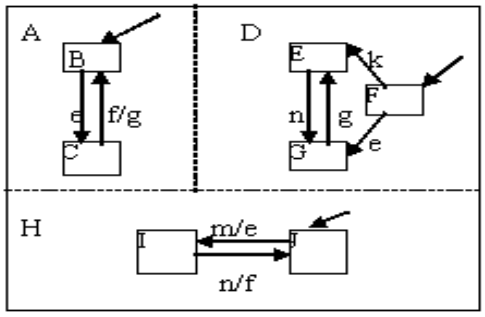


Fig. 16 State Configuration

A *state configuration* is a maximal set of states that the system can be in simultaneously. In other words, state configuration is a set of mutual parallel states, i.e., if the system is in state B, it must be in one of the substates of *superstate* D and one of substates of *superstate* H in Figure 16. The set {A, B, D, E, H, I} is maximum and consistent, and it is a *state configuration*.

The *initial state configuration* $\boldsymbol{C_0}$ is defined to be the start configuration of Statechart. It is the configuration at the time when the Statechart is first initiated by an outside event. In Fig. 16, the set {A, B, D, F, H, J} is the *initial state configuration*.

A *system configuration* $\boldsymbol{C}$ is a set of all possible *state configurations* of a Statechart. In Fig. 17, C={(G, A, H, B, I, C), (G, D, H, B, I, C), (G, A, H, E, I, C), (G, D, H, E, I, C), (G, A, H, B, I, F), (G, D, H, B, I, F), (G, A, H, E, I, F), (G, D, H, E, I, F)} is *system configuration.*

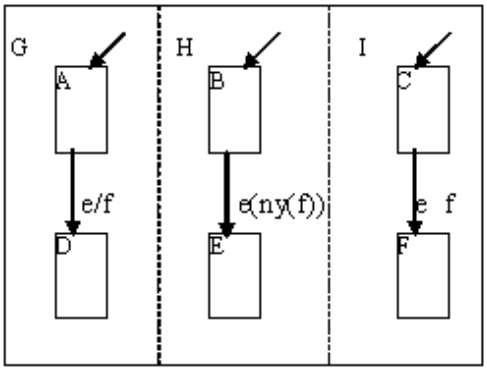


Fig. 17 System Configuration

B. Special Events and Conditions

Besides the features described in Sections 2 and 3, Statechart also includes compound events and conditions, and shared variables that can be assigned to and tested.

If *e* and *f* are events, so are $e \wedge f$ and $e \vee f$. Similar to events, if *P* and *Q* are conditions then $P \wedge Q$ and $P \vee Q$ are conditions too. Assignments include *c:=true, c:=false*, where *c* is a condition, and $v := s$ for a variable *v* and an appropriate algebraic expression s. For expressions t and s, $t = s$ and $t < s$ are conditions.

In addition, there are several special kinds of events and conditions:

- event *en(s)* occurs upon entering state *s;*
- event *ex(s)* occurs upon exiting state *s;*
- *in(s)* is the corresponding condition for *en(s)* and *ex(s);*
- event *ch(v)* occurs when *v* changes value where *v* is a variable;
- event *tr(c)* occurs when condition *c* changes from false to true;
- event *fs(c)* occurs when condition *c* changes from true to false;
- condition *ny(e)* stands for that event *e* has *not-yet* occurred;
- condition *cr(c)* stands for the current value of condition *c*;
- condition *cr(v)* refers to the current value of a variable or expression *v.*

Conditions *ny(e), cr(c),* and *cr(v)* refer only to what is happening inside the currently evaluated *chain reaction* that will be discussed later.

C. Action and Chain-reaction

In section 2, the reaction part is expressed only by the system changing its internal state configuration to incoming or sensed events and conditions. All the transitions do not contain any outputs. *Parallel components* can synchronize only through common events and can affect each other only through *in(s)* special condition. The real subtlety of the way Statechart models concurrence is in their output events. Statechart can be viewed as an extension of *Mealy machines,* such that it has the ability to generate events and change the values of conditions. These output events denoted by /s are called *actions* to be attached optionally to the labels of transitions. The enriched transition labeling is the form *e(p)/s* where *e* is the event triggering the transition, *p,* the condition that guides the transition, and *s*, the action to be carried out upon the transition.

However in contrast to conventional *Mealy machines*, an *action* appearing along a transition in a Statechart is not merely sent to the "outside world" as an output. The action typically will affect the behavior of the Statechart itself in *parallel components*. This is achieved by a simple *broadcast* mechanism just as the occurrence of an external event that causes transitions in all *parallel components*.

In Fig. 16, the initial configuration is {A, B, D, F, H, J}. When event *m* occurs and a transition labeled *m/e* is taken, the transition leads from state J to state I. In H, the action *e* is immediately activated and is regarded as an internal event. The *e* event further triggers two transitions labeled *e* in A and D. These transitions will lead from state F to state G and from state B to state C. This is a *chain reaction* of length 2. The next *state configuration* should be {A, C, D, G, H, I}. If now an external event *n* occurs, the transition labeled *n/f* in H is taken. The action *f* is immediately activated and *f* event causes transition labeled *f/g* in A to be taken. The event *g* is activated, and it further triggers transition *g* in D. The new *state configuration* is {A,B,D,E,H,J} now by virtue of a *chain reaction* of length 3.

D. Step

In Statechart, the system reaction at some instant is composed of the set of transitions taken at that instant and the set of events generated when these transitions are taken. Informally, the behavior of a *superstate* is the execution of its substates. A *parallel state* behaves either as one of its *AND components* or according to the transitions between the states. In Statechart, this kind dynamic behavior is based on the notation of *step*.

A *step* is a set of consistent transitions that are structurally relevant to a given *state configuration*. A *step* is initiated when an external event arrives at the Statechart boarder. That initial event will cause a cascade of subsequent internal events. These internal events are results of taking the steps' transitions and executing the actions associated with these transitions. A *step* is completed when no more internal events are generated or there are no more transitions triggered by the events that were generated. In Statechart, it is assumed that a *step* is completed before another external event arrives.

Let $\boldsymbol{T}$ denote a *step*. When *step* $\boldsymbol{T}$ is taken, all the states must remain consistent. All the transitions participating in a step $\boldsymbol{T}$ are taken simultaneously. In Statechart, a *step* can be further defined as a sequence of *micro steps*.

A *micro step* is a set of internal transitions. *Micro steps* capture the internal order in which the simultaneous transitions and actions of a single *step* are carried out. To protect *chain reactions* from incoming external events (coming from outside of the state machine), *micro steps* should be performed before any new steps. A *micro state configuration* is defined as the abstract system status between *micro steps*.

Given a *system configuration* $\boldsymbol{C}$, the system reaction at some instance is composed of a sequence of *micro steps*. The first *micro step* is defined as a set of transitions that occur at the current *state configuration*. In Fig. 16, if the system is in *state configuration* {A, C, D,G,H,I} and an external event *f* occurs, then the event *f* will be triggered and transition

labeled *f/g* is taken. The action *g* is immediately activated. Special condition *ny(f*) is updated to reflect that event *f* has been executed in the present *micro step*. The first *micro step* in this example results in *micro state configuration* {A,B,D,G,H,I}. In Fig. 16, the second *micro-step* will execute the transition labeled *g* and then reach the stable and consistent *configuration* {A,B,D,E,H,I}. A sequence of *micro steps* terminates. A *step* is accomplished, and the next *state configuration* is {A, B, D, E, H, I}.

E. Non-termination

Statechart employs a broadcasting mechanism to handle concurrence in the system. One part generates an event by an action and all other parts sense it and act in response if so specified. Upon sensing such an event, another component might generate a new event, causing yet other events to be generated. This chain reaction could keep going. The events generated by actions may cause non-termination problems. Cycles, like that of Fig. 18, have to be dealt with, presumably rendering them undefined.

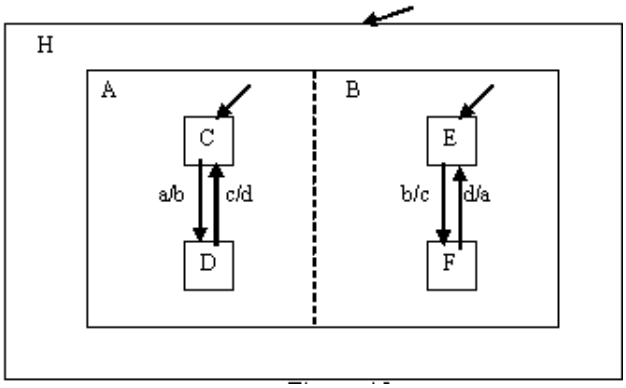


Fig. 18 Non-termination

In Fig. 18, the initial configuration is {H,A,C,B,E}. If event a occurs, transition labeled a/b will be taken and action b is generated immediately. The system is in {H,A,D,B,E}. State E senses action b and transition labeled b/c in state B is taken. Action c is generated and the system is in {H,A,D,B,F}. State D then senses action c and transition labeled c/d in state A is taken. Action d is generated and the system is in {H,A,C,B,F}. State F senses action d. Transition labeled d/a in state B is taken and action a is generated. The system is back in {H,A,C,B,E}. State C senses action a and the transition labeled a/b will be taken. The cycle will start again and never terminate.

V. Discussion and Related Work

Heral [3, 4] has accomplished his attempts to overcome the drawbacks of FSM such as flat, sequential, uneconomical transitions and states by using the Statechart. Statechart can be used to represent a large and complex state machine by adding the features of depth, parallel, and broadcast-communication.

However there are several drawbacks of the original Statechart. The order of the transitions taken place is important. The Statechart shows structure non-determinism caused by the freedom of selecting subsets in micro-steps and the uncertainty of selecting concurrent events. One of the most important properties of any real-time system is the time constraint that should be clearly indicated in the requirement specification. Although Statechart provides *timeout* feature and *time bound*, these features are not well defined and are not sufficient to represent the critical time requirements.

Recently several approaches to requirement specification for process-control are proposed and based on the Statechart. The Irvine Safety Research Group [13] borrows the notions of *superstate, parallel state, broadcast communication, state arrays*, and *conditional connectives* from Statechart. The group enhances the Statechart by adding interface descriptions and directed communication between state machines in its RSML(Requirements State Machine Language.) RSML has some unique syntactic and semantic features that are developed to enhance readability, reviewability, analyzability, and the ability to handle complex systems. Furthermore [7] defines the formal semantics of RSML and describes an automated approach to analyze an RSML specification for completeness and consistency.

Real Time Logic (RTL) [9] is a first order predicate logic invented for reasoning about time properties of real-time systems. *Modechart* [12] uses the concept of mode from the work of Parners [7] and borrows from Statechart the very appearing compact representation of large state machines. The main contribution of *Modechart* is in providing a semantics that explicitly deals with the absolute timing of events and avoids some of the potential semantic anomalies of Statechart. The translation of a *Modechart* specification into RTL will result in a hierarchy organization of the RTL assertions.

STATEMATE uses Statechart, Activity Charts, and Structure Charts for the specification analysis and documentation of large and complex reactive systems. It enables a user to prepare, analyze, and debug diagrammatic descriptions of system under development from three interrelated points of view, capturing structure, functionality, and behavior. In addition to the use of Statechart, the main novelty of STATEMATE is that it "understands" the entire descriptions perfectly. STATEMATE is able to analyze them for crucial dynamic properties, to carry out rigorous executions and simulations of the described system. STATEMATE will create running code automatically.

Besides Statechart based requirement specification, other notable approaches are used. Petri nets [15, 17] are used for visually specifying parallel/distributed software. They are graphical and precise, and they are heavily event-driven allowing maximum concurrence. Duration calculus is a real-time interval logic, where predicates define duration of states. Papelis and Casavant [17] proposed a specification which is a set of formulas in duration calculus. SCR (Software Cost Reduction) was introduced more than a decade ago [6, 7] and has been extended recently [14, 18, 19]. It is a state-based approach using an assortment of tabular notation to define state transitions and output variables. SCR requirements are intuitive, easy to write and change, and scalable to large systems.

References

[1] Ambler, Scott W., The Agile Scaling Model (ASM): Adapting Agile Methods for Complex Environments**"** , IBM, December 2009.

[2] Boehn, B.W., "Software Engineering Economics," Englewood Cliffs, NJ, Prentice-Hall, 1981
[3] Harel, D., "Statecharts: a visual formalism for complex systems," Science of Computer Programming, vol. 8 1987
[4] Harel, D., A Pnueli, J.P. Schmidt, and R.Sherman, "On the formal semantics of Statecharts (Extended Abstract)," Proceedings of the Second Symposium, Logic in Computer science, Ithaca, N.Y. 1987
[5] Harel, D., H. Lachover, A Naamad, A Pnueli, M. Politi, R. Sherman, A Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: a working environment for the development of complex reactive systems," IEEE Transactions on Software Engineering, vol. 16, No. 4, April 1990
[6] Heninger, K., "Specifying software requirements for complex systems: new techniques and their application," IEEE Transactions on Software Engineering, SE-6(1), January 1980
[7] Heninger, K., D.Parnas, J. Bhore, and J. Kallander , "Software requirements for the A-7e air craft," technical Report 9586, NRL, Washington D.C. December 1983
[8] Heimdahl, M.P.E. and N. Leveson, "Completeness and consistency in Hierarchical state-based requirements," IEEE Transactions on Software Engineering, vol. 22, No. 6, June 1996
[9] Jahanian, F. and A. Mok, "Safety analysis of timing properties in real time systems," IEEE Transactions on Software Engineering, vol. SE-12, September 1986
[10] Jahanian, F. R., Lee, and A. Mok, "Semantics of Modechart in Real Time Logic," Proceedings of the 21st Hawaii International Conference on system Science, January 1988
[11] Jahanian, F., A. Mok, and D. Stuart, "Formal specification of real-time systems," Department of Computer Science, TR-88-25, University of Texas, June 1988
[12] Jahanian, F. and A. Mok, "Modechart: A specification language for real-time systems," IEEE Transactions on Software Engineering, vol. 20, N0. 12, December 1994
[13] Leveson, N.G., M.P.E. Heimdahl, H.Hildreth, and J.D. Reese, "Requirements specification for process-control systems," IEEE Transactions on Software Engineering, vol. 20, No. 9, September 1994
[14] Parnas, D. and J. Madey, "Functional documentation for computer systems engineering," Version 2, Technical Report CRL 237, Telecommunications Research Institution of Ontatrio (TRIO), McMaster University , Hamilton, Ont., 1991
[15] Papelis, Y.E. and T.L. Casavant, "Specification and analysis of parallel/distributed software and systems by Petri Nets with transition enabling function," IEEE Transactions on Software Engineering, vol. 18, No. 3, March 1992
[16] Rawn, P. and K.M. Hansen, "Specifying and verifying requirements of real-time systems," ," IEEE Transactions on Software Engineering, vol. 19, No. 1, January 1993
[17] Reising, W. ,"Petri Nets: an introduction," Spriger Varlag, Berlin, 1985
[18] Van Schouwen, A.J., "The A-7 requirements model: reexamination for real-time and an application for monitoring systems," Technical Report TR 900-276. Queen's University Kingston, Ont., 1990
[19] Van Schouwen, A.J., D.L. Parnas, and J. Madey, "Documentation of requirements for computer systems," Proceedings of RE's 93, Requirements Symposium, San Diego, January 1993

Wendy Wenhui Zhang was born in Shanghai, China. She got her M.S. and Ph.D. in computer science from University of Houston, Texas, United States.
She is full professor of Computer Science & IT Department in Southeastern Louisiana University, Louisiana, United State. Her research interests are spatial database, high performance computing, and hyperspectral remote sensing.
She is a member of ACM and IEEE. She was awarded summer research fellowships from NASA and Naval Research Lab dring 2002-2009.

Theresa Beaubouef earned the B.S. in Computer Science from Louisiana State University and the M. S. and Ph.D. degrees from Tulane University in New Orleans, Louisiana in 1992. Dr. Beaubouef has worked as a computer scientist for the U.S. Navy and its contractors, and as Assistant Professor at Xavier University in New Orleans. She is currently a professor at Southeastern Louisiana University.
Her research interests include uncertainty in databases, data mining, spatial databases, artificial intelligence, and computer science education. Dr. Beaubouef is also interested in scientific computing and mathematical applications and formal modeling of processes.

Dr Huilin Ye is an Associate Professor at School of Electrical Engineering and Computer Science, University of Newcastle, Australia. Her main research interest is in the area of software engineering, including software product line engineering, object-oriented software development, software reusability, and software library systems etc. She is the leader of Software Engineering Research Group at University of Newcastle and currently leading an Australian Research Council funded project in feature model based software product line engineering. Prior to her academic career she had been a senior software engineer and system analyst in software engineering industry for more than 10 years.