2010 Asia Pacific Software Engineering Conference

Bridging the Gap Between Fault Trees and UML State Machine Diagrams for Safety Analysis

HyeonJeong Kim,$^{\alpha, \beta, 1, *}$ W. Eric Wong,$^{\beta, 2}$ Vidroha Debroy,$^{\beta, 3}$ DooHwan Bae$^{\alpha, 4}$

$^{\alpha}$CS division, EECS Department
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
{^{1}hjkim, ^{4}bae}@se.kaist.ac.kr

$^{\beta}$Department of Computer Science
University of Texas at Dallas
Richardson, Texas, USA
{^{2}ewong, ^{3}vxd024000}@utdallas.edu

Abstract

Poorly designed software systems are one of main causes of accidents in safety-critical systems, and thus, the importance of safety analysis for software has greatly increased over the recent years. Software safety can be improved by analyzing both its desired and undesired behaviors, and this in turn requires expressive power such that both can be modeled. However, there is a considerable gap between modeling methods for desired and undesired behaviors. Therefore, we propose a method to bridge the gap between fault trees (for undesired behavior) and UML state machine diagrams (for desired behavior). More specifically, we present rules and algorithms that facilitate the transformation of a hazard (in the context of fault trees) to a UML state machine diagram. We illustrate our proposed approach via an example on a microwave-oven system. Our proposed transformation can help engineers identify how the hazards may occur, thereby allowing them to prevent the hazard from occurring.

Keywords - safety analysis; fault tree analysis (FTA); UML state machine diagrams; automatic transformation rules

I. INTRODUCTION

Software is often responsible for controlling the behavior of electro-mechanical components and the interactions among the components in systems [8], and since most accidents occur due to control-related issues, software directly or indirectly affects the hazards of safety-critical systems [4,8]. These hazards in turn can lead to losses that may span life, property, and even the environment. Thus, safety analysis for software is highly important and must be carried out [7,8].

Fault Trees (FTs) are widely employed in hazard analysis because they are a simple, visual, and standardized notation used to state safety requirements [6,12]. Since a FT is suitable for identifying the causes of hazards and their relationships, it has been widely used in describing hazards related to hardware, which are usually composed of failure modes of hardware [6,12]. Unlike hardware, software contributes to a hazard due to an incorrect combination or sequencing of normal behaviors [1,2,14]. Therefore, safety analysis for software hazards should be with respect to the normal behavior of the system [1,2]. However, FTs are not appropriate to describe, and analyze normal behaviors (that are highly relevant to the causes of hazards) for safety analysis.

UML state machine diagrams are appropriate for specifying the discrete behavior of various subsystems [3,9]. They may also be adapted to describe the identified hazards in the FTs, which can help in the understanding of hazards with respect to system behaviors. For this purpose, UML state machine diagrams should be consistent with FTs. However, it is error prone to manually construct a UML state machine diagram from a FT. It is also difficult to formally and/or manually analyze the diagram for safety, since the diagram can have indirect paths to causes of the hazards due to its depth (i.e., a composite state can own its sub-states) and orthogonality (i.e., regions in a state are independent of each other).

Therefore, the major research goal of this paper is to develop an algorithm to transform hazards of fault tree into UML state machine diagram for safety analysis, in accordance with software behavior. This software behavior, for our purposes, is described in UML state machine notation, and hereafter we refer to it as the *original state machine (diagram)*. We develop three steps for transforming the hazards: identifying the types of primary events in the fault tree related to software behavior, developing the rules to map the primary events and gates (which are components of fault tree) to UML state machine notation, and extracting important information from the original state machine diagram that deals with properties (e.g., hierarchy or orthogonality) of the diagram. The output of our transformation process is also a state machine (diagram), that is hereafter, referred to as the *transformed state machine* (*diagram*). The contributions of the paper are as follows.

- Rules are proposed that provide the foundation of the automatic transformation from fault tree to state machine diagram for describing hazards
- Each transformed state machine diagram captures both explicit and implicit causes that trigger a hazard with respect to normal behavior

The rest of this paper is organized as follows. Section II describes the fundamentals of fault tree analysis and UML state machine diagrams. Section III then describes the methodology, while Section IV explains our approach via an illustrative example. Subsequently, Section V overviews related work, followed by Section VI which discusses the

*Corresponding author. HyeonJeong Kim is a PhD student at Korea Advanced Institute of Science and Technology and is visiting the University of Texas at Dallas as an exchange PhD student under the supervision of Professor W. Eric Wong.

identified threats to validity. Finally, Section VII presents conclusions and a discussion on future work.

II. Background

A. *Fault Tree Analysis (FTA)*

FTA can be described as an analytical technique, whereby an undesired event of the system is specified, and the system is then analyzed in the context of its environment and operation, to find all credible ways in which the undesired event (called a *top event*) can occur [6,12]. A typical fault tree is composed of a number of symbols which are classified in three categories: primary events, intermediate events, and logic gates. Primary and intermediate events represent causes of the top event which are abnormal conditions that may lead to a reduction in, or loss of, the capability of a functional unit to perform a required function. Primary events of a fault tree are those events, which cannot be further developed. The gates serve to permit or inhibit the passage of fault logic up the tree. The gates show the relationships of events needed for the occurrence of a 'higher' event. The 'higher' event is the 'output' of the gate; the 'lower' events are the 'inputs' to the gates. The gate symbol denotes the type of relationship of the input events required for the output event. There are three basic types of fault tree gates: the *OR* gate, the *AND* gate, and the *NOT* gate [6,11,12]. *PAND* is a special case of *AND* gate. Also a *XOR* gate is a combination of three basic gates, so we assume it is replaced with the basic gates in fault tree.

A fault tree is reduced to a logically equivalent form, showing a specific combination of primary events, which is called a Prime Implicant (PI) [12]. PIs may cause the top event and cannot be reduced in number.

Any fault tree consists of a finite number of PIs that are unique for its top event. The PI expression for the top event can be written in the general form,

$$T = P_1 + P_2 + \ldots + P_k$$

where T is the top event and each P_i is a Prime Implicant. Thus, T occurs if any P_i is evaluated to true. Also,

$$P_i = X_1 \bullet X_2 \bullet \ldots \bullet X_n$$

where X_1, X_2, etc., are primary events. Thus, P_i is evaluated to true only if $X_1, X_2, \ldots, X_n$ all evaluate to true, i.e., all primary events must occur.

B. *UML State machine diagrams*

Since UML state machine diagrams are used to specify the behavior of various model elements, they are an appropriate notation to describe safety-critical system [3,9]. While a state machine diagram allows for depth and orthogonality, which makes it convenient to both use and understand, at the same time this makes it equally difficult and inconvenient to analyze. A formal definition makes the analysis easier, and so we define the state machine diagram using formal notation. Due to concerns such as space limitations, and in order to maintain our focus, we omit certain aspects of state machine diagrams that we do not require, such as redefined elements or connection points. Additionally, we do not explain each element in great depth, and direct readers to [9] for more detailed definitions. We start with the definition of an *event*. We adhere to the convention that a large cased letter such as 'X' refers to a set, and one of its elements is represented by a small cased 'x', i.e., $x \in X$.

Def. 1. (Event) An event $e \in E$ is an occurrence of signal or operation (method) of a state machine diagram.

Def. 2. (Arithmetic Operation) An arithmetic operation $o \in O$ is an assignment statement that may include an arithmetic operator (e.g. +, –, /, *).

Def. 3. (Behavior) A behavior $b \in B$ is an atomic element which generates an event or does an arithmetic operation with attributes (or parameters) of a state machine diagram. (i.e., $B \subseteq (E \cup O)$)

Def. 4. (Transition label) A transition label $tl \in TL$ is 3-tuple $<E_{tl}, g_{tl}, A_{tl}>$, where E_{tl} is a set of trigger events that may fire the transition tl, g_{tl} is a Boolean expression (called a guard condition) which should be satisfied when one event in E_{tl} occurs, A_{tl} ($\subset B$) is a set of optional behaviors (called actions) which are executed after firing the transition tl.

A transition label in the diagram uses a notation '$E_{tl}[g_{tl}]/A_{tl}$.' The portion before '/' describes what is required for firing the transition. Once at least one event in E_{tl} occurs and g_{tl} is satisfied, the transition is fired. Since the inclusion of E_{tl}, g_{tl}, and A_{tl} in the notation are optional, transition labels allows for several variations in their forms: empty, only consisting of events, guard conditions, or actions, or combinations of them.

Def. 5. (Pseudo State) A pseudo state ps is either *initial*, *fork*, *join*, or *choice*.

An *initial* pseudo state is used to describe the default state in composite state or state machine diagram. The other pseudo states are used to connect multiple transition paths to or from states [9]. A *fork* pseudo state is used to split an incoming transition in to two or more transitions, and a *join* pseudo state is used to merge two or more incoming transitions. Finally, a *choice* pseudo state serves to describe a conditional branch.

Def. 6. (State) A state $s \in S$ is 4-tuple $<En_s, D_s, Ex_s, R_s>$, where En_s, D_s, Ex_s are sets of entry, doActivity, and exit behaviors of the state s respectively (i.e., $En_s \subset B$, $D_s \subset B$, $Ex_s \subset B$), $R_s \subset R$ (see Def. 8) is a set of regions of state s.

The number of regions determines whether the state is simple, composite, or orthogonal composite. A simple, composite, or orthogonal composite state has zero, at least one, or more than one region respectively. En, D, and Ex are a set of optional behaviors that are executed whenever the state is entered, while being in the state, and whenever the state is exited, respectively.

Def. 7. (Transition) A transition $t = <v_t^s, tl_t, v_t^t>$ is a relation in $V \times TL \times V$, where V is a finite set of vertices, where each vertex is either Pseudo State ps or State s (i.e., $V \subseteq (PS \cup S)$), v_t^s is a source vertex of t, v_t^t is a target vertex of t, and $tl \in TL$ is the transition label of t.

Def. 8. (Region) A region $r \in R$ is a 2-tuple $<V_r, T_r>$, where $V_r \subset V$ is a finite set of vertices in region r, and T_r is a finite set of transitions in region r.

Def. 9. (State Machine) A state machine *sm* is a set of regions, $sm = <R_{sm}>$ where $R_{sm} \subset R$ is a set of regions and $|R_{sm}| \geq 1$.

A state or state machine can own its regions and these regions are annotated with unique identifiers such as *s1*, *sm1*, so the entire set of regions (R) may be divided into subsets (R_{sm1}, R_{sm2}, R_{s1}, and R_{s2}) that are mutually disjoint, and each of them can identify its owner. Also, since regions can own states and states can own regions, they serve as to link a state machine and its states, or a composite state and its sub-states. For example, if *sm1* owns *r1* and *r2*, and *r1* owns *s1*, *s2*, *s3*, *t1*, and *t2*; then *sm1* is a parent of *r1* and *r2*, and *sm1* is a grandparent of *s1*, *s2*, and *s3*. We can thus, retrieve the hierarchical relationship between states, or between states and state machine diagrams.

III. Transformation of the Hazards from Prime Implicants of Fault Trees to State Machines

We propose three steps for transforming hazard scenarios: (1) identifying the types of primary events in fault tree related to software behavior, (2) developing the rules to transform the primary events and gates in fault trees to UML state machine diagrams, and (3) extracting the information from original state machine diagrams.

A. Identification of types of primary events in a fault tree

Fault trees should be defined with respect to the specification of the system [6,12]. The specification itself can be based on requirements documents, or design documents. In the paper, we assume that the engineer defines the fault tree based on a state machine diagram (which we refer to as the original state machine diagram), which means each primary event in the fault tree consists of what the diagram has. We assume that each primary event is specified as follows:

Def. 10. (Primary event) A primary event $pe \in PE$ is defined with a notation '*sm.element*', where *sm* is a target state machine and *element* can be the name or label of elements to be analyzed for safety.

In the first step, we identify the type of primary events using text matching with the original state machine diagram and put a 'type' tag to each primary event. We first match '*sm*' with the name of the state machine, and then search '*element*' with a label or name of element in the state machine. Primary events can be interpreted as multiple elements in the original state machine, which means that primary events can be developed further. For example, a primary event '*sm1.a=b+c*' can be mapped into both *En* and *A*. It should be developed to two lower primary events with *OR* gate. All the mapped elements can be the causes of hazard which hazard analysis method should deal with. We should, therefore, adjust the fault tree if a primary event is interpreted as more than one element in original state machine diagram. After matching, we put a tag on every primary event, where each tag shows the type of primary event and its location in original state machine diagram. Table 1 shows the types of primary events and their corresponding tags. We can classify the types of primary events into five categories: states, transitions, operations, events, and guard conditions. State machines, regions, and pseudo states show the relationship between them and other elements (state, transition, etc).

Table 1. Types of Primary events and their Corresponding Tags

Types	**Possible tag**
State	*s<sm>element*
Transition	*t<sm, source_state_ name>. transition_label*
Operation	*en<sm, state_name>.element*
	d<sm, state_name>.element
	ex<sm, state_name>.element
	a<sm, source_state_name >. transition_label
Event	*e<sm, source_state_name>. transition_label*
	en<sm, state_name>.element
	d<sm, state_name>.element
	ex<sm, state_name>.element
	a<sm, source_state_name >. transition_label
Guard condition	*g<sm, source_state_name >. transition_label*
	data<variables>.element

Identifying states is simple because all of the elements in the same state machine should have distinct names. For example, suppose that there is a primary event '*door.open*' in fault tree. The primary event can be divided into two parts: *door* and *open*. '*door*' represents a name of state machine diagram and '*open*' describes an element of the diagram. If '*open*' is mapped to a name of state, the primary event is a state. Its tag is '*s<door>.open*.' On the other hand, identifying a transition is quite different from identifying a state since the transition label can have several variations as discussed earlier (please see Def. 4 in Section II). Thus, there is a potential problem in that a primary event for a transition may not be perfectly matched (string matched) to a state machine. To illustrate this, consider a scenario where there are three transitions:

(1)'*[a>b]*',
(2)'*receive_event()[a>b]*', and
(3)'*receive_ event()[a>b]/a=b*'

in the state machine. The first transition has only a guard condition; the second transition has a guard condition and a receiving event from the state machine diagram (the second transition occurs if the guard condition is true, and it receives the event); and the third transition is same as the second transition except that there is now an additional action. Given a primary event '*sm.receive_event()[a>b]*' in the fault tree, it can refer to the second transition (if the engineer wants a perfect matching), or either of the last two transitions (if the engineer considers all elements in state machine diagram which include (textually) the primary event). We select the latter approach to get all the possible elements in state machine diagram (even for behavior, event, or guard condition). Therefore, *sm.receive_event()[a>b]* becomes an intermediate event, and consists of an *OR* gate composed of two primary events: '*sm.receive_event()[a>b]*' and '*sm.receive_event()[a>b]/a=b*.' Describing a specific transition requires information about its state machine, source state, and transition label; thus, a transition tag is formed according to the format: '*t<sm, source_state_ name>.transition_label*.'

The arithmetic operation is one kind of behaviors as mentioned in Def. 3, so it may be mapped to element in *En*, *D*, *Ex*, and *A* (as per Def. 4 and 6). Furthermore, one operation can appear in several of these behavior sets (*En*, *D*, *Ex*, and *A*) at the same time. The types of behavior sets decide when a behavior is executed, so we distinguish the types using a tag. In case of state related behavior, a tag shows its behavior type, state machine, state name, and operation (as '*element*'). Otherwise (in case of transition related behavior - action), the tag has the same structure as a transition.

An event can be mapped to a trigger event of a transition (via the transition label, Def. 4) or behavior (Def. 3 and Def. 6). If a primary event is mapped to an event in a behavior set, its type and tag have the same mapping rules as arithmetic operations. If a given element is mapped to a trigger event of the transition, we identify the element as one of E_{tl} (in Def. 4). Its tag requires the same information as a tag of a transition.

A Boolean expression (in the primary event) can be used to represent a guard condition in a transition, or alternatively to evaluate the value of an input variable. We call the second case a 'data comparative.' Data comparatives mean Boolean expressions including comparison operators (e.g. <=, >=, >, <) to compare a variable's value. Because the change of a variable's value can be one cause of a hazard, we also need to consider all the possible ways that satisfy the causes of hazard. Whenever the change occurs, the data comparatives should be checked. Tags for data comparatives require information on the list of variables in a Boolean expression.

Since we map all the possible primary events and expand the fault tree, we need human intervention (further discussion on this is presented in Section VI) to filter out the primary events which cannot be causes of the hazard. We then extract PIs from adjusted fault tree. Since the extraction of PIs is not our focus, we use existing methods such as those in [6,12].

B. Transformation of Primary Events and Gates in Fault Trees

Each PI represents a unit to be analyzed as part of the safety analysis. However, we still need information from state machine such as incoming transitions for triggering *En,* and fault tree is not appropriate to describe it. We therefore transform, the PI to a state machine to describe how causes of hazard can occur. While original state machine diagrams focus on describing how the software system works, our transformed state machine describes how the hazard occurs. Each PI corresponds to one state machine diagram. All the primary events and gates in PIs are mapped to the elements in transformed state machine diagram.

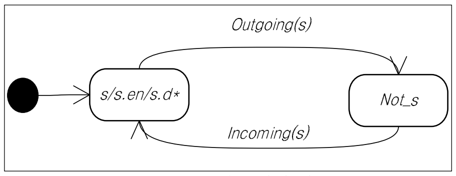
Primary events in fault trees have nine kinds of tags as shown in Table 1. Some of these tags share the same trigger conditions which are sufficient to cause a primary event to occur and so, can be grouped together. We reclassify the nine tags into four groups: (1) states, entry and doActivity of a state, (2) exit of a state, (3) transitions, events, guard conditions and actions, and (4) data comparatives. Elements in the same category have the same transformation rules.

(1) ***States, and Entry and doActivity of a state***

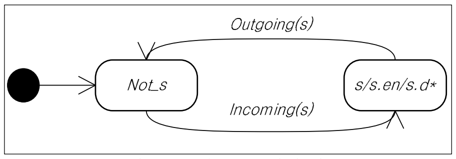
State, doActivity, and entry have same triggering conditions: firing of all incoming transitions to a given state. When a transition is fired, exit of its source state, its action, and entry and doActivity of its target state is executed in order within one unit of time. When an incoming transition to a given state is fired, entry and doActivity are executed and state becomes active. State and doActivity have the same life time; doActivity is repeatedly executed until the state becomes inactive. Thus, the three types have the same triggering conditions.

Figure 1 shows the transformed state machine diagram corresponding to a state, entry, or doActivity of the state. An *asterisk* (*) signifies the state which satisfies the cause of the hazard, and *s* corresponds to the state (it is matched to *sm.element* in its tag for state or *sm.state_name* for *d* or *en* in its tag). *Incoming(s)* refers to all transitions to *s* and *Outgoing(s)* to all transitions from *s*. Both *Incoming(s)* and *Outgoing(s)* include a set of transitions from or to other states (which excludes self-transition from *s* to *s*). *Not_s* implies all other states in same state machine diagram with state *s*.

State *s* may be a default state in original state machine diagram which has a direct transition from initial pseudo state in the outmost region. If *s* is a default state of the state machine diagram, it satisfies the cause of the hazard without firing any transitions. Figure 1 (a) shows the transformation rule of a default state *s* and Figure 1 (b) shows the transformation when state *s* is not a default state.



(a) State is a default



(b) State is not a default

Figure 1. Transformed state machine diagram for *s*, *s.en*, and *s.d*

(2) ***Exit of a state***

Exit of a state is executed when the outgoing transitions from the state are enabled (i.e., triggered event occurs and guard condition is satisfied). Even if exit behavior is included in the state, it is not executed until an outgoing transition of the state is fired. A trigger condition for an exit behavior of state is firing of outgoing transitions of a given state. Figure 2 shows the transformed state machine corresponding to exit behavior. *s* corresponds to the state including exit behavior (it is matched to *sm.state_name* in a tag for *ex*). *target(Outgoing(s))* describes a set of target states of outgoing transitions from *s*. We describe the state list in one state as per [9] and mark * on the target states. *Outgoing(target(Outgoing(s)))* are outgoing transitions from *target(Outgoing(s))*. *not_target* stands for every state in *sm* except *s* and *target(Outgoing(s))*.

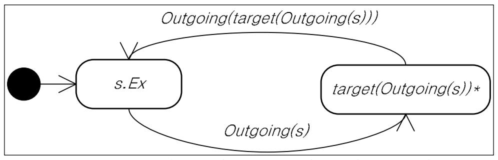


Figure 2. Transformed state machine diagram for *s.ex*

(3) *Transitions, events, guard condition, and actions*

Transitions, events, guard condition, and actions can be combined into one because they occur when the transition is fired. Figure 3 shows the transformed state machine diagram corresponding to a transition. A given transition *t*, which can be found using *sm*, *source state name*, and *transition label* in its tag) includes events, guard condition, or actions. We accept the *target(t)*, which represents a target state that is active from firing *t*, as a cause of hazard. Our concern is only occurrence of *t*, so *target(t)* is marked with *. *Not(target(t))* represents all states which do not have a given transition *t* as their incoming transition.

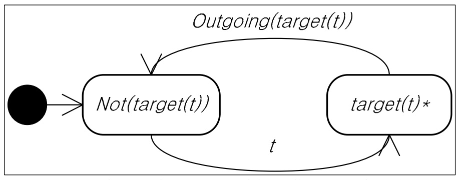


Figure 3. Transformed state machine diagram for *tl*, *e*, *g*, and *a*

(4) *Data comparatives*

Data comparatives are related to a cause of hazard related to the value of data (e.g., parameters or variables). Whenever values are changed, data comparatives should be evaluated. If the result of the evaluation is true, it satisfies the condition to be a cause of the hazard. Figure 4 describes transformed state machine diagram for data comparative '*a>b*' (described in a tag as '*element*'). *variables* in a tag represents variables (or parameters) in data comparative such as *a* and *b*. *write(variable)* returns transitions or states which define values of *variables* in their behavior. If the trigger event has each given variable as one of its parameters, the variable accepts its value from where the event is generated. Also, each given variable may take its value from result of arithmetic operation if it is located in left of assignment operator in *A*, *En*, *D*, or *Ex*. *write(variables)* returns transition, incoming transitions, state, or outgoing transitions for an assignment statement in *A*, *En*, *D*, or *Ex* respectively. *Wait* in Figure 4 stands for the state that dissatisfies the data comparative (Boolean expression) and does not have any change on *variables*. Whenever the value is changed (at least one of the transitions in *write(variable)* is enabled), it reaches a state '*check*' in Figure 4. If the data comparative (*[a>b]*) is evaluated as true, it reaches a state '*checked*', which can be a cause of the hazard and marked with *. Otherwise, it comes back to a state *Wait*.

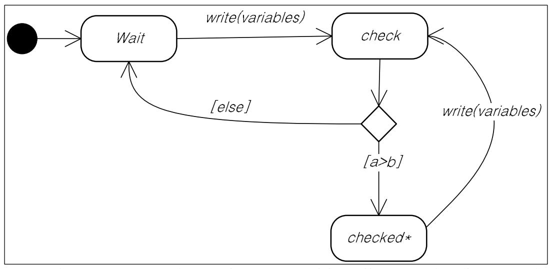


Figure 4. Transformed state machine diagram for data '*a>b*'

Each fault tree can have four gates: *AND*, *OR*, *PAND*, and *NOT* gates, where each explains the relationship between its inputs and output. Each gate of same type acts in the same manner within the transformed diagram.

(5) *AND gate*

An *AND* gate is used to show that the output occurs only if all the inputs occur [6,12]. Figure 5 describes a 2-input *AND* gate (left) and its transformed state machine diagram (right). The fault tree shows that *C* is true when both *A* and *B* occur (which is described in Boolean algebra form: *C=A•B*). All the inputs of *AND* gate should occur prior to output, so the transformed state machine starts from the inputs (the inputs come first after the initial pseudo state). A composite state after the initial pseudo state provides the regions for expanding the inputs. The input is either an intermediate event or a primary event in fault tree. If it is an intermediate event, it transforms further for its gate and inputs. Otherwise, it follows transformation rules for a primary event. Each outmost region in *AandB* has one state marked with *asterisk* (*). The states should be active at the same time, so we use a join pseudo state to merge the transitions from different orthogonal regions. Join pseudo state in Figure 5 enables the transition from state *A* and *B* to state *C* only when *A** and *B** are active at the same time. The intermediate event *C* may be used for an input of higher intermediate event, so we also mark *C* with *.

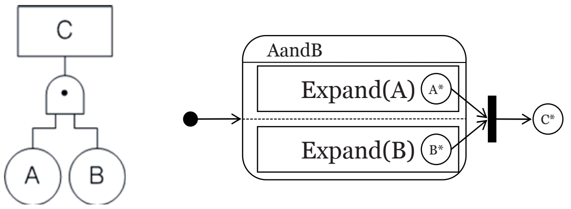


Figure 5. *AND* gate and its transformed state machine diagram

(6) *OR gate*

An *OR* gate is used to show that the output occurs only if one or more of the inputs occur [6,12]. Figure 6 describes a 2-input *OR* gate (left) and its transformed state machine diagram (right). The fault tree shows that *C* is true when either *A* or *B* occur (which is described in Boolean algebra form: $C=A+B$). The transformed state machine diagram of *OR* gate is similar to that of the *AND* gate. The difference is the connection between its inputs and output. Output *C* has direct transitions from *A** and *B**. A transition from *A** to *C* can be fired without regard to an active state in a region for expansion of *B* by [9]. Since we use a PI as a unit of the transformation and PI has only *AND* or *PAND* gates, the fault tree has only one *OR* gate which makes a connection between top event and PIs.

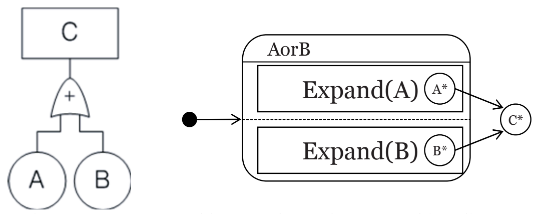


Figure 6. *OR* gate and its transformed state machine diagram

(7) *PAND gate*

A *PAND* gate means that output occurs when inputs occur in sequence [6,12]. The left portion of the input should occur before the right, i.e., inputs, which are closer to left side of a *PAND* gate, should be executed earlier. Figure 7 shows a 2-input *PAND* gate (left), and its transformed state machine diagram (right). The fault tree shows that *C* is true when *A* and *B* occur in sequence (which is described in Boolean algebra form: $C=\overline{AB}$). Unlike other gates, the composite state has only one region and each input connects in sequence with respect to order of inputs in fault tree (*A* before *B*). In the *ApandB*, the state with * in outmost region of the composite state '*Expand(A)*' has an outgoing transition to the composite state '*Expand(B)*,' and the state with * in '*Expand(B)*' has an outgoing transition to output state *C*.

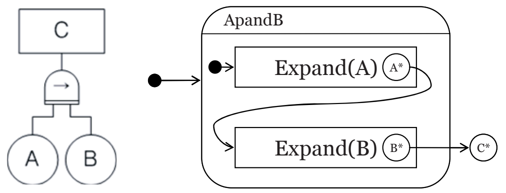
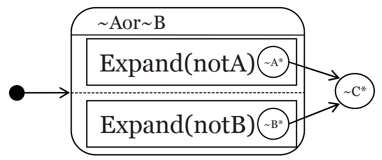


Figure 7. *PAND* gate and its transformed state machine diagram

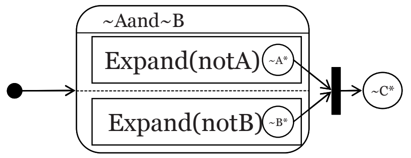
(8) *NOT gate*

A *NOT* gate means that output is a negation (or an occurrence failure) of its input [11]. Since any event (except top event) in fault tree can be an input of *NOT* gate, we should develop a method to apply the gate to all the events in fault tree. If the primary event is related to first three groups (state, exit of state, and transition), we apply *NOT* gate (or a negation) by swapping the '*' mark on the state, i.e., if one state has a '*' mark, we remove it and add it on the other state. For example, *Not_s*, *s.ex/not_target*, and *Not(target(t))* in Figure 1, Figure 2, and Figure 3 respectively have '*' marks after applying the negation. In the case of data comparatives, we reverse the comparative operator (e.g., $\leq \leftrightarrow >$, $\geq \leftrightarrow <$, $= \leftrightarrow !=$). For example, we simply negate a data comparative by replacing '$a<b$' with '$a\geq b$' in Figure 4.

For intermediate event, we can also apply the negation to swap '*' mark between output state and composite state for inputs of the gate, like a primary event. After the negation, the composite state has '*' mark instead of the output state. For instance, the output state *C* is marked with '*' in Figures 5, 6, and 7 which are pre-negation, whereas after negation the composite states '*AandB*,' '*AorB*,' and '*ApandB*' are marked with '*', and *C* would be unmarked. Since a state marked with '*' is connected to outside region, it has risk to fire a direct transition from a composite state without exploring inside the composite state. If there is no inconvenience, we prefer to delegate the negation to gate and inputs of the intermediate event. Figure 8 shows negation of an intermediate event including *AND* and *OR* gates. The negation of output (an intermediate event) is equal to negation of gate and its inputs. Negation of *AND* gate is *OR* gate and vice versa. Figure 8(a) shows negation of intermediate event ($C = A\bullet B$) including *AND* gate. Negation of *C* is represented in '$\sim C = \sim(A\bullet B) = (\sim A)+(\sim B)$.' Figure 8(b) shows negation of intermediate event ($C = A+B$) including *OR* gate. Negation of *C* is described in '$\sim C = \sim(A+B) = (\sim A)\bullet(\sim B)$.' Because all possible sequences of its inputs of *PAND* gate should be specified for negating a *PAND* gate, we prefer swapping '*' mark for *PAND* gate instead of delegating the negation to *PAND* gate and its inputs.



(a) Negation of an intermediate event which includes *AND* gate



(b) Negation of an intermediate event which includes OR gate
Figure 8. Negation of (a) *AND* and (b) *OR* gate

The *NOT* gate is also useful to collectively describe everything but a particular element. For example, suppose a region is composed of five states: *A*, *B*, *C*, *D*, and *E*. If given primary events are '*A*, *B*, *C*, or *D*,' we can replace four primary events with one primary event (*~E*). Four primary inputs are inputs of *OR* gate, and this produces four PIs instead of just one. It thus, helps to simplify and abstract both the fault tree, and the transformed model.

C. Extracting the information from state machine diagram

In this step, we extract the information from original state machine diagram such as deriving actual incoming transitions (instead on 'incoming'). The extraction is usually straightforward and explicit such as getting a source/target state of a given transition. However, elements that are related to hierarchy and orthogonality in the original diagram have implicit information which should be analyzed. For example, firing an outgoing transition from a composite state makes both the state and its sub-states inactive. These kinds of implicit transitions for the sub-states make the analysis of state machine diagram difficult, but it should still be considered. We explain the elements which have these hierarchy and orthogonality issues.

(1) *Checking a default state*

A UML state machine diagram allows depth (i.e., hierarchy), and so it might have more than one active state at the same time. Figure 9 shows the pseudo code to check whether a given state is one of the default states in state machine diagram *sm*. Only if the state *s* and its ancestor states are default states, is *true* returned and *false* otherwise. The parent of *s* is a region (see Def. 8) which includes an initial pseudo state, and its one outgoing transition whose target state is the default state in the region. If the target state is *s*, we need to extend the examination to its ancestor states. Its ancestor states are retrieved by getting its grandparent (by Def. 6, Def. 8, and Def. 9). The procedure repeats until it reaches to state machine level.

```
Boolean default(State s){
        if(localDefaultState(s)){
                element e=s.getGrandParent();
                if(type of e is state)
                        return default(e);
                else //type of e is state machine
                        return true;
        }
        else{
                return false;
        }
}
Boolean localDefaultState(State s){
        Region r=s.getParent();
        Vertex v=r.getInitialPseudoState();
        If(s==v.outgoingTransition()->target())
                return true;
        return false;
}
```

Figure 9. Pseudo code for checking if *s* is a default state

(2) *Retrieving all incoming transitions*

We should extract the incoming transitions to the state *s*, in order to fully transform the state and its behaviors. Figure 10 describes pseudo code to retrieve all incoming transitions to the state *s*. The incoming transitions can be direct or indirect. Direct incoming transitions mean they head to a given state. Indirect incoming transitions mean they make a given state active even though its explicit target is a different state.

Indirect incoming transitions can be further divided into three cases. The first case occurs when there is a transition from outside to inside of a given state. Even though a target of the transition is not a given state, it is included in a given state. If the transition is fired, both a given state and a target state become active. The second case occurs when a given state is a default state and there are incoming transitions whose target is its super-state (composite state). If an incoming transition, which terminates on the outside edge of the composite state, is fired, each default state in each region is active [9]. Incoming transitions to the composite state are also treated as implicit incoming transitions to default states in it. The third case indicates when the state is a default state and there is a transition to a state in other regions of its super-state (orthogonal composite state). Whenever an orthogonal composite state is entered, each one of its orthogonal regions is also entered, either by default or explicitly [9]. Therefore, if a transition from outside the orthogonal composite state, to a state inside of other orthogonal regions is fired, each region of the orthogonal state are entered and its default state become active.

```
Transition[] getIncomingTransitions(State s){
        Transition [] incoming;

        //direct incoming transitions
        incoming.add(s.getIncomingTransitions());

        //indirect incoming transition #1
        if(s.isComposite()==true){
                Transition t ∈ s.getParent();
                if(t->target() ∈s&&t->source() ∉ s)
                        incoming.add(t);
        }
        while(localDefaultState(s)){
                Element e=s.getGrandParent();
                //indirect incoming transitions #2
                incoming.add(s.getIncomingTransitions());

                //indirect incoming transitions #3
                Transition t ∈ e.getParent();
                if(t->target() ∈e&& t->target() ∉s.getParent()
                        && t->source() ∉e)
                        incoming.add(t);
                s=e;
        }
        return incoming;
}
```

Figure 10. Pseudo code for retrieving incoming transitions

(3) *Retrieving all outgoing transitions*

We should extract the outgoing transitions to the state *s* for transformation of all the primary events, which is similar to extracting incoming transitions. Figure 11 describes pseudo code to retrieve all outgoing transitions to the state *s*. The outgoing transitions can also be direct or indirect. Direct outgoing transitions mean that they start from a given state.

Indirect outgoing transitions mean they make a given state inactive even though its explicit source is not *s*. Indirect outgoing transitions can be further divided into three cases. The first case occurs when there is a transition from inside to outside of a given state. If the transition is fired, both the source state and the given state become inactive. Even though the source of the transition is not the given state, the source is included in the given state and the transition goes out of the given state. The second case occurs when there are outgoing transitions from the super-state of a given state (a given state is included in a source state of transition). When exiting from a composite state, the active sub-states (including a given state) are exited [9]. Outgoing transitions from the composite state are also treated as implicit outgoing transitions from all of its sub-states. The third case indicates when a given state is active, and there is an outgoing transition to a state in other regions of its super-state (orthogonal composite state). When exiting from an orthogonal composite state, each of its regions is exited [9]. Therefore, if a transition from a state inside of other orthogonal regions, to outside the orthogonal composite state is fired, each region of the orthogonal state are exited and its active states are exited.

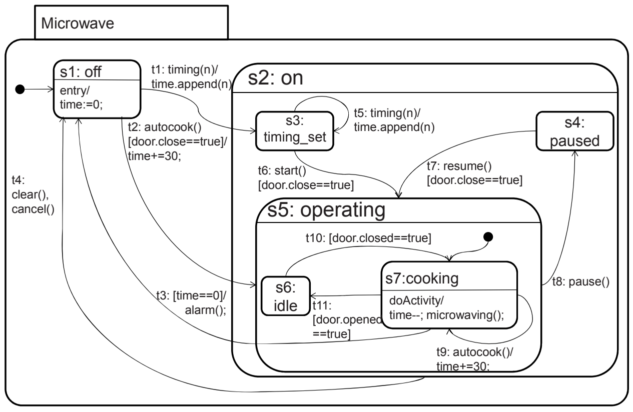
```
Transition[] getOutgoingTransitions(State s){
        Transition [] outgoing;

        //direct outgoing transitions
        outgoing.add(s.getOutgoingTransitions());

        //indirect outgoing transition #1
        if(s.isComposite()==true){
                Transition t ∈ s.getParent();
                if(t->source() ∈ s && t->target() ∉ s)
                        outgoing.add(t);
        }
        Element e=s.getGrandParent();
        while(type of e is a state){
                //indirect incoming transitions #2
                outgoing.add(e.getOutgoingTransitions());

                //indirect outgoing transitions #3
                Transition t ∈ e.getParent();
                if(t->source() ∈ e && t->target() ∉ e
                        && t->source() ∉ s.getParent())
                        outgoing.add(t);
                s=e;
                e=s.getGrandParent();
        }
        return outgoing;
}
```

Figure 11. Pseudo code for retrieving outgoing transitions

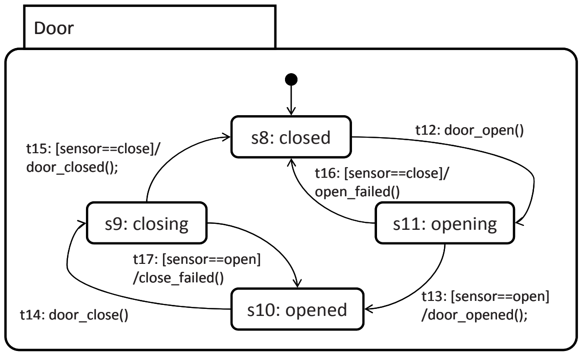
IV. An Illustrative Example

We now describe how our approach helps engineers bridge the gap between fault tree analysis and system specification using the example of a microwave oven. The software system is described as a UML state machine diagram (as shown in Figure 12) and one hazard is defined in a fault tree (as shown in Figure 13 (a)). States and transitions in the UML state machine diagram and primary events in fault tree are specified with their own identifiers (e.g., *s1*, *t1*, or *p1*). We depict two parts of a microwave oven: microwave and door. The microwave describes how the user sets the cooking time, and operates it (as per Figure 12 (a)); and door specifies how the system understands the user's control to open or close the door (as per Figure 12 (b)). The fault tree shows the hazard that involves user exposure to microwave radiation, because the microwave produces radiation without the door being closed, and the radiation may harm humans.

Figure 13 (b) shows the result after identifying the types of primary events in fault tree (which is the first step). *p1* is matched to one of doActivity in state *cooking* of state machine *microwave*, and *p2* is mapped to state *closed* of state machine *door*. If one primary event has more than one possible element in original state machine diagram, it should be developed to have all possible candidates as its input and PIs of fault tree should be retrieved again. For instance, suppose that there is another state *s4* including *micro waving()* in its entry. *p1* should be an intermediate event and it has *OR* gate and its two inputs: *p3*-'*d<microwave*, *cooking> micro-waving()*' and *p4*-'*en<microwave*, *paused> microwaving()*.' Then, the fault tree consists of one top event, one *OR* gate, and two PIs: *p3 AND ~ p2* (PI1), and *p4 AND ~p2* (PI2).

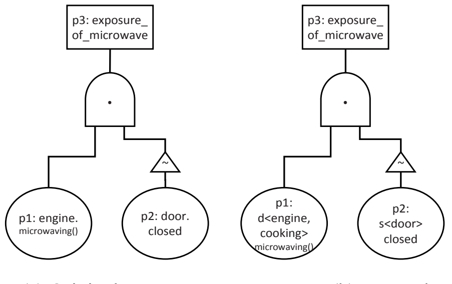


(a) State machine diagram for microwave



(b) State machine diagram for door

Figure 12. UML state machine diagram for microwave



(a) Original (b) Mapped

Figure 13. Fault tree for hazard of microwave

Figure 13 (b) is an input for the second step and Figure 14 shows a result after transforming the input. The rule for an intermediate event having *AND* gate is applied first and it decides the shape of the orthogonal composite state '*s7_d_and_not_s8*' (for inputs of the gate), join pseudo state (for AND gate), and simple state '*exposure_of_microwave*' (an output of the gate). Each region in the composite state provides space to expand each input (*s7_d* or *not_s8*). While the region *s7_d* is applied a transition rule for *s.d* (Figure 1 (b)), the region *not_s8* is applied a transition rule for *s* and a *NOT* gate (switching '*' on the states in the region).

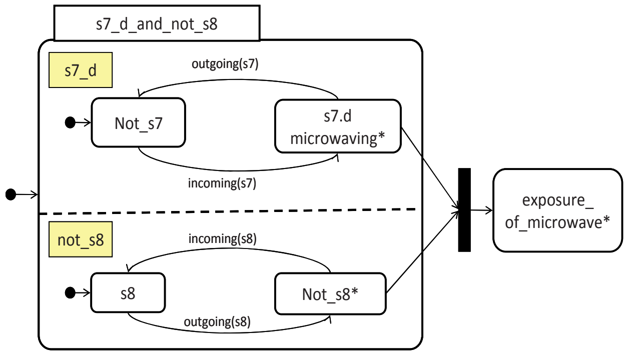


Figure 14. Transformed state machine diagram for the hazard without information from original state machine diagram

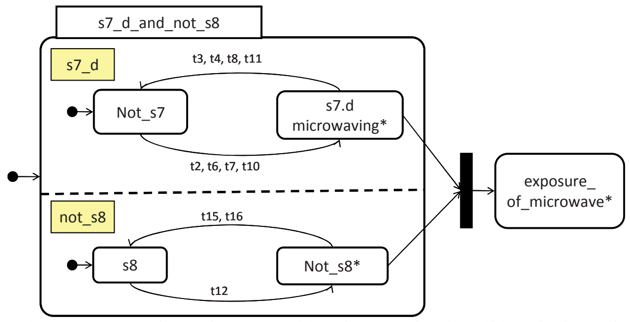


Figure 15. Transformed state machine diagram for describing the hazard with information

Figure 15 shows the result after retrieving all the transitions (the third step). Except *t10* (direct incoming), *t3* and *t11* (direct outgoing), all transitions to be derived in region *s7_d* are indirect incoming or outgoing transitions of *s7*. *s2* includes *s5* and *s5* includes *s7*, so incoming and outgoing transitions of *s2* and *s5* should be considered. They are not orthogonal composite states and *s7* is their inner most state, so there exist only the second type of indirect transitions (which start from or terminate on the outside edge of the composite state). *s5* includes *s7* as a default state; all incoming transitions (i.e., *t2*, *t6*, and *t7*) to *s5* are also treated as incoming transitions to *s7*. Since *s5* is not a default state, *s7* does not accept incoming transition to *s2* even though *s2* is super-state of *s5* and *s7*. If the outgoing transitions from *s2* (*t4*) and *s5* (*t8*) are fired, *s7* becomes inactive.

While the original state machine diagram (as shown in Figure 12) describes the normal behaviors of microwave, our transformed state machine diagram (as shown in Figure 15) focuses on describing the abnormal behaviors (hazards) of microwave, which consists of wrong combination or wrong sequences of normal behaviors. The transformed diagram depicts the required elements that are sufficient to cause the hazard to occur, and so it acts a basis to produce the test scenarios to examine if software does not have any defined hazard in the fault tree. For instance, we can combine a set of test scenarios using *s7.d*_microwaving (i.e., {*t2, t6, t7, and t10*}) and *not_s8* (i.e., {t12}) in Figure 15. As a result of combination (referring to Figure 12 and Figure 15), '*t2 → t12*,' '*t1 → t6 → t12*,' '*t2 → t8 → t7 → t12*,' and '*t2 → t11 → t10 → t12*' can be derived as test scenarios. With these test scenarios, we can find a remaining hazard in software, and thus we can prevent the hazard from occurring, for example by changing the guard condition of *t11* to '*door.closed==false*' or the triggering event of *t11* to '*door.door_open().*'

Even though engineers define the hazards using fault trees, the gap between fault tree and original state machine diagram makes it hard to use the fault tree for safety analysis. For instance, we cannot trace what triggers *p1* only with the fault tree (of Figure 13). If *t10* is a primary event in the fault tree, the original diagram is inappropriate to describe *s7*'s being active only due to firing *t10* because *s7* has other three defined incoming transitions. The transformed diagram can link this gap between the fault tree and the original diagram.

Safety-critical software may have a very complicated structure in its model, and so analyzing the large and complicated model manually is time consuming and effort intensive work. Furthermore, it is easy to make unexpected errors or miss hazards to be analyzed. Especially, dealing with the implicit information related to depth and othogonality manually make these problems worse. Thus, providing the rules and algorithm to semi-automate (if not fully automate) the transformation helps significantly to reduce effort, time, and errors in interpreting the hazards in fault tree with respect to the original state machine diagram.

V. Related Work

Several studies have tried to bridge the gap between fault tree and system modeling [1,2,5,7,10]. Kaiser provided an integrated approach to add the notion of states and events to fault trees, which are called State/Event Fault Trees (SEFTs) [1,2]. SEFT adopts the state/event concept to depict the system behavior, and the fault tree concept to describe the faults which are connected to states or events. It has a combined view for both system specification and fault. However, the engineer manually identifies the causes of fault tree and manually connects them to state or event. Constructing SEFT is very difficult, error-prone, and time consuming especially when the system is very complicated and large. Furthermore, because both system specification and hazard are defined in one model together, the engineers should require additional effort to divide a view for a specific purpose.

Some researchers have tried to combine fault tree and state-based model (Petri-net [5,7] or state machine [10]). They provided the method to convert the fault tree to state-based model. It provides simple transformation without connecting the fault with information in state-based modeling. Consequently, the converted result conveys exactly the same meaning as the fault tree. To use the transformed information, the engineer should identify and connect each cause with model element manually.

VI. THREATS TO VALIDITY

We expand the fault tree for capturing all possible causes, when we identify types of primary events in fault tree. Domain experts should get rid of false causes among primary events. If the domain experts are not involved in elimination, the normal behaviors may be treated as abnormal behaviors and the engineer may try to remove or block normal behaviors, which might lead to unexpected hazards. Therefore, we rely on the domain experts' opinion for the elimination.

In the second step, we re-classify and transform each type of primary event which are identified in the first step. Re-classification is in accordance with conditions sufficient to lead to a cause, which excludes exact time when it happens or data dependencies. For example, firing a transition is a moment, but we accept the target state after the transition is fired rather than the firing moment. Instead of adopting the moment, we separate and abstract one state (in original diagram) into two states according to possibility of being a cause of hazard. For example, if the target state is active due to other incoming transition (in transformation of transition *t*), we do not accept it as the cause because the given transition is not fired. The transformed state machine admits only the active target state after the given transition is fired.

Our example is based on a microwave-oven system, and not a complex safety-critical system. This may affect our confidence in our abilities to generalize. However, if safety-critical system is described in UML state machine diagram, regardless of high the complexity, the transformation rules and algorithm for extracting the information can be applicable because we consider the meta-structure (i.e., depth and orthogonality) of the diagram (defined in UML superstructure [9]).

VII. CONCLUSIONS AND FUTURE WORK

We develop an algorithm to transform the hazard from a fault tree to a state machine diagram, which bridges the gap between hazard analysis and system specification. It helps the engineer to develop the primary events of the fault tree by matching them with elements of state machine diagram. The algorithm provides an automatic transformation, and it deals with implicit transitions of the state machine diagram that the engineers can overlook. The resultant state machine diagram focuses on the causes of the hazard and shows the direct paths to the causes, which can potentially help to identify the test scenarios.

As far as future work is concerned, we will work on extracting test scenarios from transformed state machine diagram with larger and more complex safety critical software systems, and also work on developing a tool to support automation from transformation to extraction of test scenarios. Unlike previous work which aims to extract test scenarios [13], we can focus on undesired behavior as well as normal behavior.

ACKNOWLEDGMENT

This research is funded by the following agencies: (1) The MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-(C1090-1031-0001)), (2) The Defense Acquisition Program Administration and Agency for Defense Development under the contract, (3) The Real-world Scale Computing: BK21 Program, KAIST in 2010, and (4) The USA National Science Foundation (NSF) grant CCF-0851848.

REFERENCES

1. B. Kaiser, "Extending the Expressive Power of Fault Trees," Annual Reliability and Maintainability Symposium (RAMS), pp. 468-474, January, 2005.
2. B. Kaiser, C. Gramlich, and M. Förster, "State/event fault trees - safety analysis model for software-controlled systems," International Journal of Reliability Engineering and System Safety, Vol. 92, no. 11, pp. 1521-1537, November, 2007.
3. D. Harel, "Statecharts: A Visual Formalism For Complex Systems," Science of Computer Programming 8(3), pp.231-274, 1987.
4. E. Wong, V. Debroy, A. Surampudi, and H. Kim, "Recent Catastrophic Accidents: Investigating How Software Was Responsible," Proc. International Conference on Secure Software Integration and Reliability Improvement (SSIRI 10), Singapore, June 9-11, 2010.
5. K. Buchacker, "Combining fault trees and Petri nets to model safety-critical systems," High Performance computing. The society for computer simulation International, pp. 439 - 444, 1999.
6. M. Stamatelatos, W. Vesely, "Fault Tree Handbook with Aerospace Applications," Technical Report of NASA, August, 2002.
7. N. Leveson and J. Stolzy, "Safety analysis using Petri net," IEEE Transaction on Software Engineering, Vol. 13, No. 3, pp. 386-397, March, 1986.
8. N. Leveson, "Safeware: system safety and computers," Addison-Wesley, September, 1995.
9. Object Management Group (OMG), "OMG Unified Modeling Language (OMG UML), Superstructure," formal/09-02-02, February, 2009.
10. O. El Ariss, X. Dianxiang, E. Wong, C. Yuting, and L. Yann-Hang, "A Systematic Approach for Integrating Fault Trees into System Statecharts," Proc. 32nd Annual IEEE International Computer Software and Applications Conference, pp.120-123, 2008.
11. T. Chu and G. Apostolakis, Methods for Probabilistic Analysis of Noncoherent Fault Trees, *IEEE Transactions on Reliability*, Vol. 29, No. 5, pp. 354-360, December, 1980.
12. W. Vesely, F. Gold berg, N. Roberts, and D. Haasl, "Fault tree handbook," Technical Report NUREG-0492, U.S. Nuclear Regulatory Commission, 1981.
13. Y. Kim, H. Hong, D. Bae, and S. Cha, "Test Cases Generation from UML State Diagram", IEE Proceedings - Software, Vol. 146, No 4, pp. 187-192, August 1999.
14. Y. Oh, J. Yoo, S. Cha, and H. Son, "Software safety analysis of function block diagrams using fault trees," International Journal of Reliability Engineering and System Safety, Vol. 88, no. 3, pp. 215-228, June, 2005.