Integrating Safety Analysis With Functional Modeling

Omar El Ariss, Dianxiang Xu, *Senior Member, IEEE*, and W. Eric Wong, *Senior Member, IEEE*

***Abstract*—Functional modeling and safety analysis are two important aspects of safety-critical embedded systems. However, they are often conducted separately. In this paper, we present an approach for integrating fault-tree-based safety analysis into statechart-based functional modeling. The proposed approach uses systematic transformation steps that maintain the semantics of both the fault tree and the statechart. It also provides a set of conversion rules that transform the gates of fault trees into statechart notations. The resultant model shows how the system behaves when a failure condition occurs and acts as a basis model that ensures safety through requirement validation. Using the gas burner case study, we demonstrate the advantages of the integrated model over the use of separate models, such as the lack of ambiguities, separation of concerns, and taking the order of the occurrence of faults into consideration.**

***Index Terms*—Fault integration, fault tree analysis (FTA), software reliability, software safety, software validation, statecharts.**

I. Introduction

SAFETY refers to the lack of a state or a situation that can cause an accident, which is an unexpected occurrence of "death, injury, illness, damage to or loss of property, or environmental harm" [1]. When a system is monitored or controlled by software, a software malfunction might cause failures that can result in risks and accidents. The wide spread of software and the criticality of system safety entail that software safety needs to be addressed throughout the system development process.

Fault tree analysis (FTA) [2] is an engineering practice that is commonly used for the safety analysis of a system under development or an existing system. The construction of fault trees by safety or system analysts usually requires a deep understanding of the system and its components. The fault tree notations describe how certain behaviors of system components can combine to result in a system hazard or a failure. On the other hand, models that are used to depict system specifications concentrate on the dynamic behavior of the software and its components and on how these components interact with each other and with their environment. Statecharts [3] are a formalism that has been widely applied to the functional specification of software. Modeling with statecharts often focuses on the operational or intended behavior of the system. Therefore, there is a clear fault coverage gap by the system model. FTA can be an excellent candidate to narrow this gap.

A system model that takes failures into consideration is crucial in ensuring that safety is considered throughout the development process. It will offer the following benefits: The model will allow engineers to be knowledgeable about the undesirable conditions and system failures and to understand how the behavior of the system is affected by these failures. It will help them to understand the interaction between the software and other system components. The model will also identify the components that are responsible for the system functions that were previously identified by the hazard analysis. These components should then be given special attention in the system development process.

Fault trees and statecharts are used by engineers with different professional backgrounds. Integrating them raises several challenges. First, fault trees and statecharts are heterogeneous models where mismatches might occur due to missing (or additional) functionalities between the models or due to different naming conventions. Second, the leaf nodes of fault trees can represent the state transition, state occurrence, bounded state, etc. The interpretation of these nodes can be ambiguous. For example, does gas leak mean that the gas valve cannot close or that the valve is functioning normally but was left open for a longer duration of time which resulted in the excess of gas? Does Gas Leaks > 4 s mean that the gas should leak continuously for 4 s or that it could leak discontinuously? In addition, the interpretation is subjective—it can vary from one person to another.

The proposed integration method for FTA tries to take into consideration these problems and to deal with them by using a systematic set of transformation rules. It builds on our previous work [4] and directly transforms the FTA into a statechart notation without the use of an intermediate model (e.g., duration calculus) as is done by the previously proposed integration methods [5]–[8]. This direct transformation tries to maintain the structure of the FTA so that it is amenable to look at the statechart and clearly identify the components that represent the fault tree. In addition, the method allows the verification of whether there are some events and transitions in the fault tree that are already represented by the system statechart and allows the identification of those components that should be briefly modified and those that can be left as they are without being wrongly represented twice. The integration results in an integrated functional and safety specification (IFSS) model. In the

Manuscript received January 20, 2010; accepted July 13, 2010. Date of publication January 10, 2011; date of current version June 21, 2011. This paper was recommended by Associate Editor C. P. Nemeth.

O. El Ariss is with the Department of Computer Science, North Dakota State University, Fargo, ND 58108 USA (e-mail: omar.elariss@.ndsu.edu).

D. Xu is with the National Center for the Protection of the Financial Infrastructure, Dakota State University, Madison, SD 57042 USA (e-mail: dianxiang.xu@dsu.edu).

W. E. Wong is with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083 USA (e-mail: ewong@utdallas.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCA.2010.2093889

TABLE I
FAULT TREE GATE TYPES

Symbol	Gate	Meaning
$\wedge$	AND	The fault (parent node) occurs only when all the gate inputs (children nodes) occur.
$\overline{\wedge}$	PRIORITY AND	The fault (parent node) occurs only when all the gate inputs (children nodes) occur in a left to right order.
$\vee$	OR	The fault (parent node) occurs when at least one of the gate inputs (children nodes) occur.
⬡	INHIBIT	The fault (parent node) occurs only when the gate input occurs and the enabling condition part is true.

IFSS model, some integrated failures can help in proving that the requirements are met by the specification, whereas other failures can help in identifying inconsistencies, incompleteness, or missing constraints in the system specifications. As such, the main contribution of this paper is the set of transformation steps and conversion rules that integrate fault trees into statecharts. This allows the validation of safety requirements together with functional requirements.

The remainder of this paper is organized as follows. Section II gives some background about fault trees and statecharts. Section III introduces the transformation steps and conversion rules. The gas burner case study is discussed in Section IV. Section V reviews the related work. We conclude in Section VI.

II. BACKGROUND

A. *Fault Trees and Fault Tree Semantics*

In a safety-critical system, every major failure is represented by a fault tree. Each fault tree describes how the individual fault components combine to result in an undesirable system behavior or catastrophic failure. The root of a fault tree represents the major failure or the most general failure. As we go down the tree, the nodes represent more specific or detailed faults. Thus, a fault tree describes the catastrophic event in terms of its causal factors or faults in a hierarchical fashion.

A fault tree is composed of nodes, edges, and gates. A gate is a logical connective, whereas nodes are events that are considered as gate inputs, and edges connect nodes to gates. There are various types of gates that can be used in a fault tree, but in this paper, we limit ourselves to the set of gates that are defined by the Fault Tree Handbook [2] and that produce coherent tree structures. Table I shows these gate types with their respective semantics. The NOT gate and the exclusive OR (XOR) gate, which implicitly uses the NOT gate, are not considered in FTAs and in this paper because they introduce noncoherent trees and thus will increase the complexity of the analysis [9].

The analysis of a fault tree could be done either qualitatively or quantitatively [10]. Quantitative analysis computes the probability of the occurrence of the root node from the probability of the basic nodes. Qualitative analysis shows the set of failures that should occur together in order for the root node to occur. Although both methods bring useful information to the IFSS model, qualitative analysis is more important and is an essential part for introducing faults into the system model. Therefore, only qualitative analysis will be used.

The semantics of a fault tree can be deduced from the semantics of the root node, while the semantics of the root node is defined from the semantics of its intermediate nodes, gates, and edges. An intermediate node is defined by the semantics of its leaves, edges, and gates in the subtree where the intermediate node is considered as a root. Therefore, the semantics of a fault tree is only defined from the semantics of the edges, gates, and leaf nodes. The result of the fault tree semantics can then be represented by a Boolean expression formula [11]. The semantics of the gates is given in Table I, and the conversion rules to represent the gates in statechart notations will be given in Section IV.

B. *Statecharts*

A statechart [12] is a behavioral model that depicts the functional specifications of a system. A statechart is composed of states, transitions, and events. States represent the components and subcomponents of a system. Transitions between the states depict the system and subsystem interactions, while events and actions trigger these transitions. Although statecharts are effective in representing the dynamic behavior of a system, they lack the capability of qualitative and quantitative analyses of the behavioral properties [13].

Statecharts were introduced by Harel [12] as an extension to state machines. The aim was to represent the behavior of complex systems, such as reactive ones, in a clear and understandable form without suffering from explosion in the number of states and edges. Statecharts were not intended to be a mere specification tool but a formalism or a language that can be compiled and executed [14], [15]. Orthogonality is supported through the use of parallel states separated by dashed lines (AND-states). Communication between orthogonal parts is done through the use of actions and through the broadcasting of events. Hierarchy is supported by allowing states to encompass other states (OR-states). The initial work by Harel did not define semantics for the statecharts. As statecharts became more popular, many semantics were introduced [16]–[18].

III. INTEGRATING FAULT TREES INTO STATECHARTS

In this section, we describe how fault trees are integrated into the system model. The technique focuses on the integration of one fault tree at a time into the system statechart or the intermediate IFSS model. The result is a new IFSS model that incorporates both functional specifications and faults. The method uses a set of systematic transformation steps (see Section III-A) that concentrates on gates, where each gate with its inputs is considered and then integrated into the system statechart. In Section III-B, we introduce a set of conversion rules that maps gate representations into statechart notations. The transformation steps apply these conversion rules in order to integrate fault trees with statecharts.

This paper focuses on computer-controlled embedded systems, which consist of subsystem and control components. The statechart representation used in this paper is based on [12]. The only difference between the notations used here and those in [12] is in the representation of a superstate. This is done because the clear identification of a superstate through the use of dotted lines from regular states will help in the understanding of the

<**Safety-Requirement-Formula**> ::= <Gate>
<**Gate**> ::= "("<operand> ", " <Gate-Inputs>")"
<**Gate-Inputs**> ::= <leaf-node> ", " <leaf-node> | <leaf-node> ", " <Gate> | <leaf-node> ", " <Gate-Inputs> | <Gate> ", " <Gate-Inputs>
<**operand**> ::= $\wedge$ | $\overline{\wedge}$ | $\vee$ | ⬡ |

Fig. 1. BNF notation for the safety requirement formula.

<**leaf-node**> ::= <simple-definition> | <composite-definition>
<**composite-definition**> ::= "&, " <simple-definition> ", "<simple-definition> | <composite-definition> ", " <simple-definition>
<**simple-definition**> ::= "conditional statement" | "state occurrence" | "event" | "transition" | "elapse of time" | "threshold of some duration"

Fig. 2. BNF notation for a leaf node.

<**Semantic-Table**> ::= <simple-definition> <statechart-component> <Transitions>
<**statechart-component**> ::= "None" | <component> | <sub-component>
<**Transitions**> ::= {<transition>}
<**transition**> ::= <state-occurrence> [<state-nonoccurrence>]

Fig. 3. BNF notation for the semantics table.

conversion rules. In addition, in order to prevent any confusion with the term basic event in safety analysis terminology, the term event used in the subsequent sections will follow the software engineering terminology.

A. Transformation Steps

The integration process is applied to one fault tree at a time. Each individual failure (fault tree) is converted into a statechart representation and then integrated with the system behavior. The conversion is done by transforming individual gates (with their inputs) into statechart components. The statechart representation of the failure is composed of a set of orthogonal gate components. The statechart representation of the gate that triggers the root node will be the main or controller component for the statechart representation of the fault tree. This component will keep track of the occurrence of faults (leaf nodes) and the failure (the root). The rest of the orthogonal statechart components will represent individual faults.

The proposed method consists of four steps for integrating one fault tree into a system statechart. In step 1, the semantics of the fault tree is deduced. Steps 2 and 3 deal with the mismatches between the two heterogeneous models and gather additional information that will be used by the last step. In step 2, the syntactical differences between the two models are considered, while step 3 deals with the semantic differences of the two models. The last step uses the additional information that was derived by steps 2 and 3 in order to merge the failure with the system specification. Each step will be discussed in the following sections.

Step 1—Deduce a Safety Requirement Formula (Boolean Formula) From the Fault Tree: The semantics of a fault tree can be depicted through a formula that only shows the root node and how this root node can be reached through a combination of gates and leaf nodes. The Backus-Naur form (BNF) notation is shown in Fig. 1.

Step 2—Define Primitive Failure Conditions (Leaf Nodes) in the Fault Tree: Leaf nodes (basic events) are highly dependent, during the analysis of a failure, on the chosen scope and resolution of a fault tree [11]. The scope of the analysis defines which system components and faults to focus on, while the resolution defines the failure components that the analysis should stop on. These failure components are considered as basic events in a fault tree. Leaf nodes might not stand for the basic cause of a failure but match the level of detail needed for the analysis of a particular system. As a result, the level of detail for a particular system in a statechart and a fault tree representation might be different.

In order to resolve the ambiguity of the two models, we use the term "simple definition" when a leaf node can be expressed by the statechart notations shown in Fig. 2.

When a leaf node has the same level of detail as that of a statechart, it can be expressed as a simple definition; otherwise, it is composed of more than one simple definition.

To deduce the semantics table, we decompose the safety requirement formula into simple definitions. This is to check every leaf node to see whether it has a simple definition. If not, then decompose it into simple definitions. The decomposition will result in an AND gate whose inputs are simple definitions that define the leaf node. Consider a leaf node stating that a door should be open for 5 s. It does not have a simple definition because it combines two simple definitions, a state occurrence (the occurrence of an open state in the door component), and an elapse of time (5 s). It can be decomposed into the following:

$$\text{Door open for } 5 \text{ s} = \text{Door open} \wedge \text{elapse of } 5 \text{ s}.$$

Finally, modify the safety requirement formula (deduced in step 1) to reflect the changes made.

Step 3—Match the Semantics of the Statechart and the Fault Tree by Deducing a Semantics Table: Some of the leaf nodes might refer to hardware components and cannot be represented by the specifications of the software. Other leaf nodes, such as normal events which cause a fault, might already be represented in the statechart. This step helps in identifying whether the simple definitions in the requirement formula cannot be represented, are already represented, or can be represented by the system statechart. Here, functionalities are associated on a component basis. The need for a safety analyst might be required to clarify semantic mismatches such as which statechart component matches the component responsible for the leaf node.

The semantics table is a table that shows the equivalent interpretation, if available, of the system statechart for every simple definition in the safety requirement formula (from step 2). In a statechart, the main emphasis is on the states and their transitions, while in a fault tree, the emphasis is on the events and their occurrences. Then, when building the semantics table, the focus will be on the transitions. Each row has three fields as shown in Fig. 3: a simple definition, a statechart component, and equivalent transitions.

A leaf node as an input to a gate can be considered as true sometimes but false in other times. Representing the leaf node in the semantics table only as a transition from a state to another due to the occurrence of the event is not enough. Another

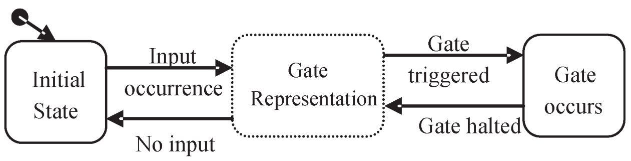


Fig. 4. Statechart notation for a general gate.

transition is needed to represent the leaf node when it becomes false (the change in input occurrence from being true to being false). This transition is indicated in the BNF notation as "state-nonoccurrence."

The requirements (or leaf nodes) that are not implemented by the system should be included as assumptions. Implemented requirements should be part of the safety commitments. Safety requirements are met when the assumptions are ensured not to occur and the system meets its safety commitments.

Step 4—Transformation of Gates in the Fault Tree: The last step transforms the fault tree into a statechart representation and then merges this failure with the system specification. This is done through the following:

1) Construct the new statechart by working on the formula from left to right starting with the first operand, then the next one, and so on. When the operand is located, work on the operand and its inputs according to the conversion rule that represents the gate as a statechart notation. Work recursively if the inputs contain any other operands.
2) If the gate inputs (simple definitions) have an equivalent statechart representation in the semantics table, then use these equivalent statechart representations during the construction of the statechart notation for the operand.
3) Add the new representations of the fault tree as orthogonal components to the original statechart.
4) Modify the controller component of the system by adding a transition, where the event that triggers this transition is the same action event that is produced by the statechart representation of the fault tree.

B. Conversion Rules

Gate transformation (see Section III-A4) considers every gate in a fault tree one at a time and converts it into a statechart notation. The main focus here is to introduce conversion rules that convert a gate with its inputs into statechart notations. Fig. 4 shows a general statechart notation of a gate. The *Initial State* is the state before the occurrence of the gate and any of its inputs. The *Gate Representation* is a superstate, represented in the figure as a dotted state, which represents the gate and its interaction with its inputs. The next sections will describe the specific notation for this superstate for every gate that is going to be represented. The *Gate occurs* state is reached when the criteria for the gate inputs are met.

In the following sections, conversion rules will cover all of the gates in Table I except for the inhibit gate. This is because the inhibit gate can be represented as an AND gate of two inputs [6], and thus, the conversion rule for the AND gate can be used instead. Two additional notations needed to represent two types of simple definition, an elapse of time and a counter, are also given a statechart representation. The conversion rules will be

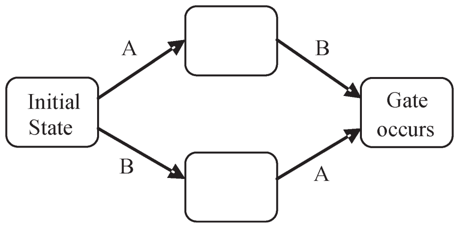


Fig. 5. AND gate representation using method 1.

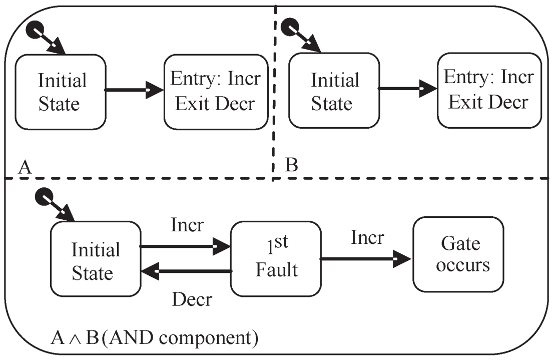


Fig. 6. AND gate representation using method 2.

further understood through their application in the gas burner case study (see Section IV).

Rule 1 (AND Gate): An AND gate can either be represented by using the following:

1) Method 1: The AND gate will be transformed into a set of transition states. This method can be applied to AND gates with a limited number of inputs. As the number of inputs increases, the representation becomes more difficult to use. In addition, method 1 does not have a general or fixed form of representation as method 2, so as the number of inputs changes, the representation of method 1 changes. For an AND gate with two inputs A and B, the representation, as shown in Fig. 5, is clear and easy to apply. This ease of use brings with it a drawback if one or more of the inputs are already represented in the system statechart. In this case, applying this method will add on to the existing statechart notations or might even accidently duplicate their representation without noticing that they have already been represented. This causes difficulties in differentiating between the statechart representations which stand for system functionalities and the statechart representations which stand for faults. Our intentions are to maintain as much as possible the structure of both models so that it is possible to look at the statechart and clearly identify the components that represent the fault tree. Therefore, method 2, through the use of orthogonality, slightly changes the existing statechart components without adding semantic ambiguities.
2) Method 2: The AND gate will be transformed into a set of orthogonal states. Using this method, the AND gate statechart notation can be seen as two parts. The first part represents the gate inputs, while the second part represents the AND gate. The first part includes an orthogonal component for every gate input. The second part, the AND component, includes an orthogonal component that

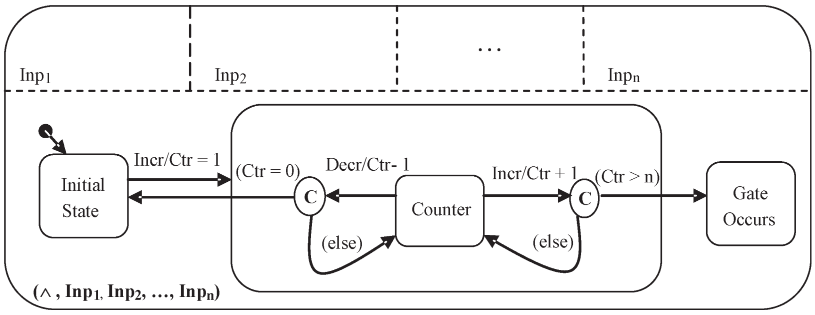


Fig. 7. General representation of an AND gate with n inputs.

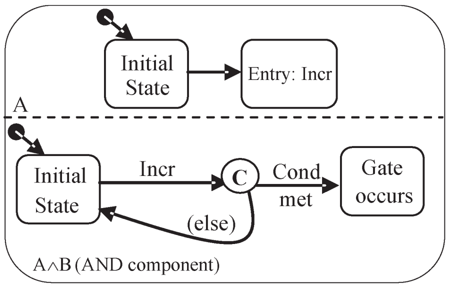


Fig. 8. AND gate representation with conditional connective.

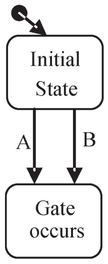


Fig. 9. OR gate representation.

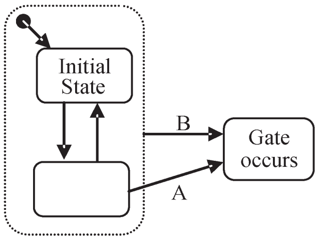


Fig. 10. OR gate representation (only B is an event).

combines all the stand-alone inputs. This AND component shows the interaction of the inputs together. An example of an AND gate with two inputs A and B is shown in Fig. 6.

It can be seen that, for every gate input, an orthogonal state is formed. The event *Incr* indicates the transition to a state where one of the inputs (either A or B) is now present, while a *Decr* event indicates the loss of availability of one of these inputs. Therefore, there is a need for two consecutive *Incr* events in order for an AND gate with two inputs to be triggered.

Method 2 has a general or fixed form of representation independent of the number of inputs. One drawback is that the representation uses individual states to represent each fault occurrence as an *Incr* or *Decr* event occurs. Therefore, as the number of AND gate inputs increases, the number of states in the AND component also increases. An alternative way is through the use of a counter notation to represent the fault occurrence states. Fig. 7 shows a general AND gate representation of n inputs. The representation still needs to include an orthogonal component for each of the n inputs, but the occurrence of these inputs is now represented as a counter state. This counter state exits in two cases: Either the counter *Ctr* is equal to n or is decremented to zero. When the Ctr reaches the value n, then all the n inputs of the AND gate have occurred and a transition to the *Gate occurs* state happens. The *Ctr* is decremented to zero when all the gate inputs cease to occur.

For an AND gate of two inputs, if one input is a bounded state or a conditional connective, then there is no need to have an orthogonal component to represent this input. This input can be directly represented in the AND component. Fig. 8 shows an example where input B is a conditional connective.

*Rule 2 (*OR *Gate):* The OR gate representation is composed of a set of transitions (one transition for each input) from the Initial State to the *Gate occurs* state. When gate inputs are not events, the only difference in the representation is that an additional orthogonal component will be added for each gate input. Then, the transitions will be triggered by these orthogonal components. For an OR gate of two inputs A and B, the transition to the *Gate occurs* state happens if either A, B, or both A and B occur. In the statechart representation, the occurrence of both A and B together is not represented because no additional information is added. Fig. 9 shows the statechart representation of an OR gate of two event inputs.

Another example is an OR gate of two inputs A and B where input A is not an event but a state occurrence. In the case where one of the two gate inputs is not an event, we can represent the OR gate in two ways. We can either represent the OR gate as in Fig. 9 and have an orthogonal component that triggers A or use the representation in Fig. 10. The only difference from the representation in Fig. 9 is that, rather than using an orthogonal component to represent A, this component is directly integrated with the *Initial State*. This integration will result in a superstate. The representation of the A component is application dependent. If A occurs, then a transition from the A component to the *Gate occurs* state will happen. If B occurs at any time, an immediate transition from any current substate in the superstate to the *Gate occurs* state will happen.

Rule 3 (Priority AND *Gate):* The priority AND gate occurs if all the gate inputs happen in a specific order. Therefore, the

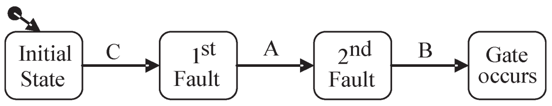


Fig. 11. Statechart notation representing a priority AND gate.

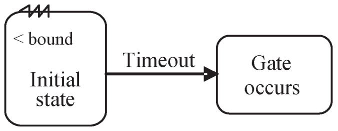


Fig. 12. Statechart notation representing an elapse of time.

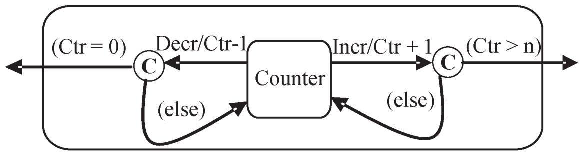


Fig. 13. Statechart representation of a counter.

representation should keep track of the targeted sequence of gate inputs. When all the inputs stand for events, the priority AND is represented as a set of consecutive states and transitions with the following conditions: 1) The number of states is less than the number of gate inputs by one; 2) the number of transitions is equal to the number of inputs; and 3) the transitions should follow the same order of the gate inputs.

An example is a priority AND gate with three inputs A, B, and C where the input sequence is C followed by A and finally followed by B. Fig. 11 shows the representation of this gate.

In the case where the inputs are not events, then orthogonal states will be used to trigger the nonevent gate inputs.

Rule 4 (Elapse of Time): Statecharts are a synchronous language [3], [18] that is suitable to model real-time, reactive, and mixed software–hardware systems. The statechart language follows the synchrony hypothesis [19] which states that reactions should take no time at all and should be instantaneous. Therefore, the execution of statechart transactions takes zero time. In this case, continuous time can be modeled through the use of time-outs [16]. In our statechart notations, we use the bounded state and time-out event features from the work of Harel [12]. An elapse of time can be represented, as shown in Fig. 12, using a bounded state. When the time exceeds the bounded time, a time-out is triggered, and a transition from the *Initial State* to the *Gate occurs* state happens.

Rule 5 (Counter): A general representation of a counter is shown in Fig. 13. The counter is represented by a state, two conditional entrances, a counter variable, and two events that increment and decrement the counter variable. Every time an increment or decrement event occurs, represented in the figure as *Incr* and *Decr*, respectively, the counter variable will be increased or decreased by one. The conditional entrances are used to check whether the counter variable has reached its bounds. Therefore, if the counter variable, represented in the figure as *Ctr*, is decremented to zero, then it will exit the counter state. The same thing occurs when the *Ctr* reaches the desired count or number, represented in the figure as n. The representation of the counter could be changed in accordance to the way that it will be used. An example of an adaptation of a counter

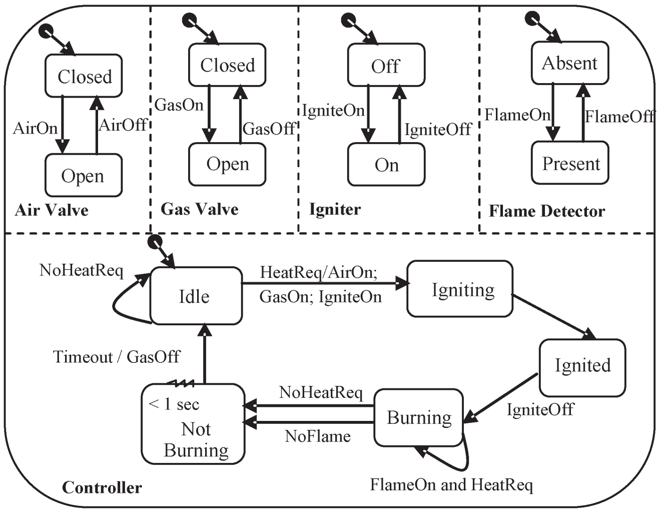


Fig. 14. Statechart for a gas burner.

that differs in its statechart representation is a counter that only gets incremented but not decremented.

IV. Case Study

This section demonstrates the usefulness and importance of our approach. We first describe the integration of one failure into the gas burner system. Then, we explain how our approach improves the analysis and validation of the underlying system. We conclude this section with some lessons learned.

A. Gas Burner

In this section, we give a detailed explanation of the integration process of a fault tree into the system statechart. Fig. 14, which is based on [5], shows the statechart representation of a computer-controlled gas burner. The functionality of a gas burner is to produce heat through the consumption of gas. The system is composed of the following components: a gas valve (responsible for the supply of gas), an air valve (responsible for air), an igniter (responsible for the supply of flame), a flame detector (monitoring the state of the flame), and a controller (monitoring the heat request process).

The requirements of a gas burner [20] can be summarized as follows: 1) At all times, the gas concentration in the surroundings of the gas burner should not exceed a certain threshold; 2) when heat is requested, the gas burner should be functioning correctly and producing heat; and 3) when heat is not requested, the gas burner should not be working or producing heat.

Fig. 15, which is based on [6], shows a failure in the gas burner. Fire is caused either through a short circuit in the cables or due to an ignition attempt while there is an excess of gas and air is present. The concentration of gas is considered excess based on the assumptions of the gas burner process [20].

Applying the Transformation Steps: The steps needed to transform and integrate the fault tree of Fig. 15 into the system statechart are as follows:

Step 1: Deduce a safety requirement formula from the fault tree. Here, the fire catastrophe (which is the root node) will

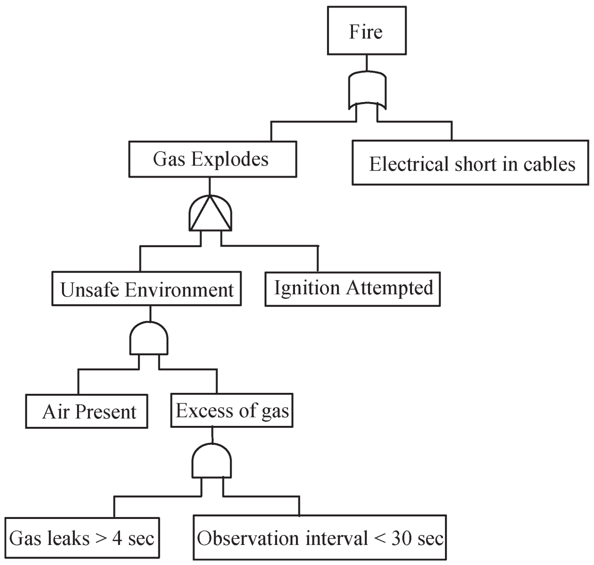


Fig. 15. Fault tree for a fire occurrence.

occur only through the combination of the leaf nodes with the gates as shown in the following:

$$\mathbf{Fire} = (\vee, (\overline{\wedge}, (\wedge, \text{Air\_Present}, (\wedge, \text{Gas\_Leaks} > 4\text{ s}, \text{Observation\_Interval} < 30\text{ s})), \text{Ignition\_Attempted}), \text{Electrical\_Short}).$$

Step 2: Check the leaf nodes of the derived formula if they are all simple definitions. In this case, all of the leaf nodes in the safety formula have a simple definition except for Gas_Leaks > 4 s, which needs to be decomposed into a "state occurrence" and an "elapse of time" (following Fig. 2)

$$\mathbf{Gas\_Leaks} > 4\text{ s} = (\wedge, \text{Gas\_Present}, \text{threshold} > 4\text{ s}).$$

That means that, in order for the gas to leak, an excess of gas should be present. In other words, gas should be present for a duration or a threshold of more than 4 s. Therefore, the safety requirement formula is now as follows:

$$\mathbf{Fire} = (\vee, (\overline{\wedge}, (\wedge, \text{Air\_Present}, (\wedge, (\wedge, \text{Gas\_Present}, \text{threshold} > 4\text{ s}), \text{Observation\_Interval} < 30\text{ s})), \text{Ignition\_Attempted}), \text{Electrical\_Short}).$$

Step 3: From the statechart in Fig. 14 and the fault tree in Fig. 15, we deduce the library of semantics, shown in Table II.

This table will help in the transformation from the fault tree notations to an equivalent statechart representation. The table is built by checking every simple definition at a time and then identifying the component in the statechart that might be responsible for this simple definition or event. The next step is to check if this component contains a transition that has an event with a similar meaning to the simple definition. If there is one, then this transition "state-occurrence" with "state-nonoccurrence" (if present) will be added to the equivalent transition column. Let us consider Air_Present, the first leaf node in the formula. Checking the statechart in Fig. 14 for a component that has the functionality to represent this leaf node will result in the *AirValve* statechart component. Checking the transitions of this component will show that the transition caused by the event *AirOn* is equivalent to the Air_Present leaf node and will cause the state of air to be true, while the event *AirOff* will cause the state of air to stop or be false.

TABLE II
SEMANTICS TABLE

Simple Definition	Statechart Component	Equivalent Transition
Air Present	Air Valve	(AirValve.Closed, AirOn, AirValve.Open) (AirValve.Open, AirOff, AirValve.Closed)
Gas Present	Gas Valve	(GasValve.Closed, GasOn, GasValve.Open) (GasValve.Open, GasOff, GasValve.Closed)
Threshold > 4 sec	Gas Valve	None
Observation Interval < 30 sec	Gas Valve	None
Ignition Attempted	Igniter	(Igniter.Off, IgnitionOn, Igniter.On) (Igniter.On, IgnitionOff, Igniter.Off)
Electrical Shortcut	None	None

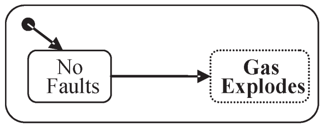


Fig. 16. Partial formation of the OR gate.

Step 4: Only one of the requirements of the fault tree (Electrical short in the cable leaf node) cannot be implemented. This leaf node will be assumed to be implementable just for OR gate representation purposes. Therefore, there is no need for assumptions. The next thing to do is to start integration by working on the formula from left to right beginning with the first operand, then the next operand, and so on. Therefore, the first thing to construct is the OR gate with its two inputs. The formula is

$$\mathbf{Fire} = (\vee, \text{Gas Explodes}, \text{Electrical short in cables}).$$

Following the rule for the OR gate as in Fig. 10, *Gas Explodes* is integrated with the initial state as shown in Fig. 16.

The presence of *Gas Explodes* is not checked in the semantics table because it is an intermediate (not leaf) node. The next step is to check the *Electrical short in cables*, which is a leaf node and located in the semantics table. It has no semantic interpretation in the library because it is hardware related. It is interpreted as an event occurrence that results in a transition from the superstate (the initial state with the integration of *Gas Explodes*) to the fire state, as can be seen in Fig. 17. The figure also shows that the fire state can be reached by either a transition from the *Gas Explodes* state (the occurrence of the *Gas Explodes* intermediate node) or a transition from any state in the superstate by the occurrence of a shortcut event. This is exactly what the fault tree in Fig. 15 indicates.

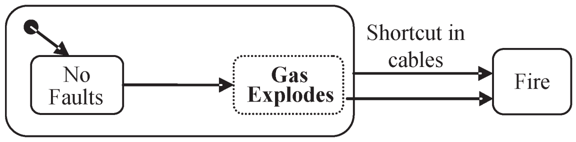


Fig. 17. OR gate as a statechart.

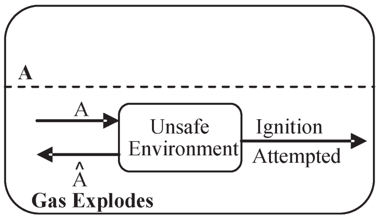


Fig. 18. Applying transformation rule to Gas Explodes.

What is left for the OR statechart in Fig. 17 is to transform the *Gas Explodes* node into a statechart, where *Gas Explodes* has the following Boolean formula:

$$\begin{aligned}&\textbf{Gas Explodes}\\&\quad = (\overline{\wedge}, (\wedge, \text{Air\_Present},\\&\qquad (\wedge, (\wedge, \text{Gas\_Present}, \text{threshold} > 4\ \text{s}),\\&\qquad \text{Observation\_Interval} < 30\ \text{s})),\\&\qquad \text{Ignition Attempted}).\end{aligned}$$

The next operand is a priority AND gate of two inputs. *Gas Explodes* now becomes

$$\begin{aligned}&\textbf{Gas Explodes}\\&\quad = (\overline{\wedge}, \text{Unsafe Environment}, \text{Ignition Attempted})\\&\quad = (\overline{\wedge}, \text{A}, \text{B})\end{aligned}$$

where

A $= (\wedge, \text{Air\_Present}, (\wedge, (\wedge, \text{Gas\_Present}, \text{threshold} > 4\ \text{s}), \text{Observation\_Interval} < 30\ \text{s}));$

B $=$ Ignition Attempted.

The priority AND gate has two inputs, where A is not an event but B (as in the semantics table) is an event. Following the priority AND gate rule, A is now represented as shown in Fig. 18.

To represent B, we check the semantics table, which indicates that *Ignition Attempted* (B) has an equivalent statechart representation in the gas burner statechart, and the *Igniter* statechart component responsible for the presence of ignition should be modified. The modification is done by changing the *Igniter* transition to integrate B as an input in the priority AND gate. That means that, whenever B is present, the Fire statechart component in Fig. 20 should change its current state to reflect this change. We are not concerned with the ignition being stopped because, when there is an *Unsafe Environment* and an ignition is attempted, the failure (fire) will occur. There is no need to modify the *IgnitionOff* event. Fig. 19 shows the modification done to the *Igniter* component (the addition of the *Fire.Incr* action).

In order for the priority AND gate to be triggered, A should occur first, then followed by B. Now, the integration of A with the initial state as indicated before will change the statechart representation of the fault tree from Figs. 17–20.

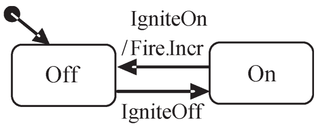


Fig. 19. Modified Igniter component (B).

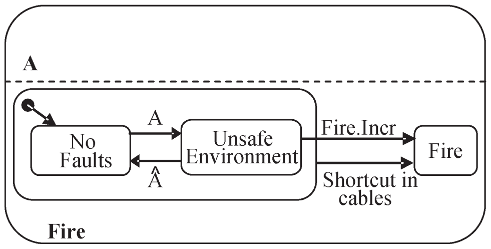


Fig. 20. Partial statechart for the fault tree in Fig. 15.

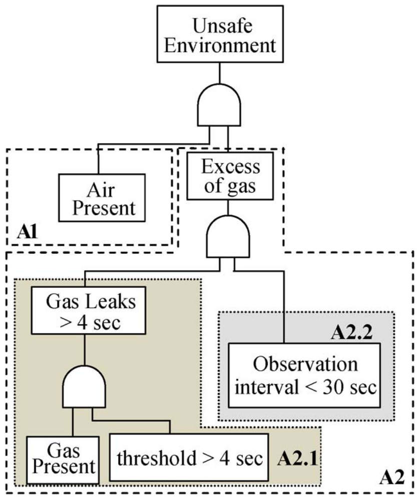


Fig. 21. Fault tree representation of A.

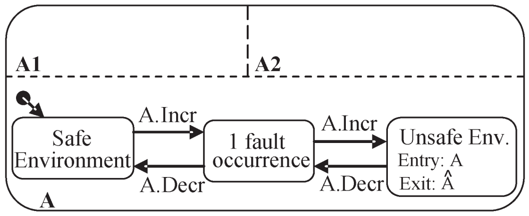


Fig. 22. Partial statechart for the fault tree in Fig. 21.

The next step is to work on A (the next operand and its inputs) and then transform it into a statechart notation. Here, the operand is an AND gate with two inputs. The fault tree representation of A is shown in Fig. 21, where A1 and A2 represent the two inputs for the AND gate.

Following the conversion rule for an AND gate and using method 2, the orthogonal state A of Fig. 20 will be transformed into that of Fig. 22. For every input, an orthogonal component is formed. *A.Incr* indicates the transition to a state where one of the inputs A1 or A2 is now present. *A.Decr* indicates the loss of availability of one of these inputs. Therefore, two consecutive *A.Incr* events are needed for A to occur.

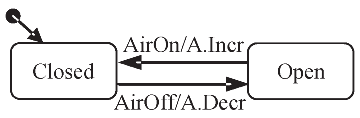


Fig. 23. Modified Air Valve component (A1).

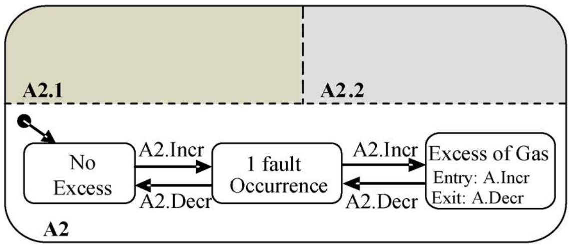


Fig. 24. Applying transformation rule 2 to A2.

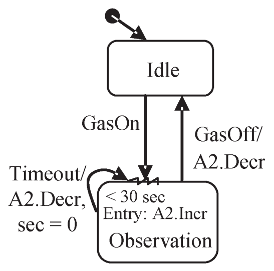
The next step is to represent A1 and A2. A1 is a leaf node. The semantics table shows that *Air Present* (A1) has an equivalent statechart representation, and the *Air Valve* statechart component responsible for the presence of air should be modified. The modification is done by changing the *Air Valve* transition to integrate A1 as an input in the AND gate. That means that, whenever A1 is present or ceases to be present, the A statechart component in Fig. 22 should change its current state to reflect this change. Fig. 23 shows these modifications (additions of the *A.Incr* and *A.Decr* actions).

Now, we consider A2. Its statechart representation depends on how its respective fault tree representation is interpreted. One interpretation is that, when the gas leaks, the leak should be continuous. That means that the gas should leak in a continuous manner for more than 4 s during the 30-s duration. This definition does not consider the fact that gas could leak for 2 s, stop leaking for a certain duration, and then leak again for 3 s during the 30-s duration. An alternative is to consider discontinuous gas leaks, which will be used in the statechart representation. Here, A2 is an AND gate with two inputs, where one of the inputs is an AND gate with two inputs

$$\mathbf{A2} = (\wedge, (\wedge, \text{Gas\_Present}, \text{threshold} > 4\text{ s}), \text{Observation\_Interval} < 30\text{ s}).$$

Fig. 24 shows a partial statechart representation of A2 using method 2 of the first conversion rule. A2 is an AND gate of two inputs, where each input is represented as an orthogonal component. The colored orthogonal components represent the same colored input nodes in Fig. 21. The occurrence of any of the A2 inputs will trigger an *A2.Incr* event, while the loss of availability of any of these inputs will trigger an *A2.Decr* event. The occurrence of both inputs will trigger this AND gate and thus will cause the occurrence of the A2 input in Fig. 22. This can be seen in Fig. 24 where the occurrence of two *A2.Incr* events will cause a transition to an *Excess of Gas* state. Upon entering this state, an *A.Incr* event will be generated and thus will cause a transition in the A component in Fig. 22.

The A2 gate has two inputs, where one of these inputs is a leaf node (A2.2) while the other is an AND gate of two inputs. Starting with the leaf node (simple definition), the semantics table shows that the $Observation\ Interval < 30$ s is not represented by the gas burner statechart. The table also shows



Observation Interval (30 sec)

Fig. 25. Representation of A2.2.

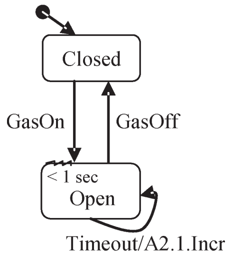


Fig. 26. Modified Gas Valve component.

that the Gas Valve component has the functionality to represent this node. This leaf node is a case of an elapse of time simple definition. Rule 4 will be used in order to represent this node. Fig. 25 shows the statechart representation for A2.2. A bounded state is used to keep track of the observation interval, and a time-out event will be triggered whenever the recorded time exceeds the bounded time. The *Idle* state indicates that there is no need to keep track of the observation interval when the gas valve is closed.

Gas Valve is the statechart component responsible for the inputs of the AND gate of A2.1. Looking at the semantics table, the first input has an equivalent statechart transition, while the second does not have any equivalent notation. Therefore, the leaf node *Gas Present* has an equivalent statechart representation in the *Gas Valve* statechart. To incorporate the leaf *Gas Present* in the A2.1 AND component, the Gas Valve component is modified as shown in Fig. 26.

The only modification to the *Gas Valve* component is changing the Open state into a 1-s bounded state that will increase the threshold by one (through the use of the *A2.1.Incr* action) for every second that the gas is being released.

As for the $Threshold > 4$ s node, the semantics table shows that the *Gas Valve* component has the functionality to represent this node but there is no equivalent statechart representation. This leaf node is responsible for keeping track of the gas leakage. Here, we only consider the case of the discontinuous gas leakage. The $Threshold > 4$ s leaf node cannot be represented using an elapse of time (rule 4) because this representation only keeps track of the continuous time. The counter representation (rule 5) can model the discontinuous gas leakage and will therefore be used instead of a bounded state. As it is the case that A2.1 is an AND gate of two inputs and the $Threshold > 4$ s node is a counter (conditional connective), then this leaf node can be directly included in the A2.1 AND component without the need for an additional orthogonal component. Therefore, instead of using a state labeled *1 fault occurrence*, this state is

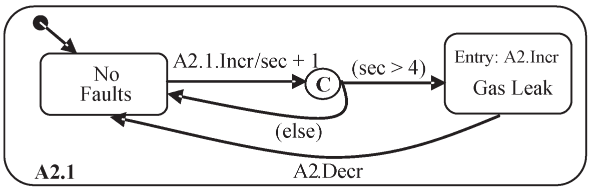


Fig. 27. Representation of A2.1.

replaced with the representation of the $Threshold > 4$ s. The A2.1 component representation is shown in Fig. 27. The leaf node is represented as a counter (with a variable called sec) that keeps track of how many seconds in which the gas is still open. When the variable exceeds 4 s, then a transition to a *Gas Leak* state occurs. The counter is cleared (the sec variable is set to zero) every time the observation interval in Fig. 25 times out.

What is left is to integrate the newly constructed statechart components into the gas burner statechart. Fig. 28 shows the complete representation of the gas burner statechart with the fire failure. The new statechart components are added as orthogonal parts. The *controller* statechart component should show that a fire might occur. This is done by adding a new state to the *controller* where the transition will be triggered when the gray-colored state in the Fire orthogonal statechart component is reached, and a *Start_Fire* action is emitted.

B. Analysis

The fault tree in Fig. 15 shows the configuration of the faults that will cause a fire failure in the gas burner. Still, the figure does not really help in identifying how these faults are linked to the gas burner specification model. The fault tree does not cover all the faults that might cause the failure but only considers the fault modes that are necessary for the safety analysis of the gas burner system. Mostly, the focus of the FTA will be on the system and its components and not on the software that controls the system. This is why it is difficult to relate the failure in Fig. 15 to the gas burner statechart or the system functionalities when the gas burner is not functioning properly.

This also implies that the fault tree is not mainly intended for the safety analysis of software due to the underlying difference between software and hardware, although there are some common safety concerns between the two systems and the software that controls them. As a result, taking safety into consideration through the use of two separate models, the statechart for the correct behavior of the software and fault trees for system failures will raise some difficulties.

1) The fault tree is mostly hardware or system specific, and some faults might not be found in the system specification. For instance, the leaf node Electrical short in cables is a hardware error and has no match in the gas burner statechart.
2) Ambiguities: The leaf node might be interpreted by software engineers in different ways due to the lack of their understanding of the system functionalities. For instance, in the case of the gas burner, the Gas Leaks $>$ 4 s leaf node has different interpretations. Does gas leak mean that the gas valve cannot close after it is opened or that the valve is functioning correctly but was left open for a duration of time which resulted in an excess of gas? In addition, should the leak be continuous for 4 s, or can it leak for a couple of seconds, function normally for a while, and then leak for an additional 2 s? Moreover, it is not quite easy for software engineers, with their understanding of only the correct behavior of the system, to locate the functional component or subcomponents that are responsible for the two leaf nodes Observation Interval $<$ 30 s and Gas Leaks $>$ 4 s. Therefore, their interpretation of those two leaf nodes might be incomplete or incorrect. For example, the Gas Leaks $>$ 4 s leaf node is not easily related to the gas burner statechart in Fig. 14, and its representation depends on the representation of the Observation Interval $<$ 30 s leaf node.
3) Order of occurrence: Should the order of the occurrence of faults be considered? When should the order of the occurrence of faults be sequential (one fault occurring after the other, where the order might or might not be important), and when should faults occur at the same time so that the parent node can occur? For instance, in the case of the gas burner, should the presence of air, the excess of gas, and the ignition attempted occur one after the other, or should all of them be present at the same time?
4) Statecharts only emphasize the functional behavior of the system. Thus, software engineers only understand the system from its intended behavior and do not clearly understand the consequences of one or more malfunctioning components on the system behavior. For instance, what will happen to the gas burner if the gas valve does not close when it should close? One of the requirements for the gas burner states that the gas concentration should not exceed a certain threshold, and therefore, the failure violates this requirement. Yet, this requirement cannot be clearly identified and understood in the gas burner statechart. It is not possible from the statechart to identify the gas concentration of a burner or what should be the threshold for this gas concentration. Furthermore, both the requirement and the statechart representation do not indicate the outcomes and safety concerns for having a gas burner that operates in an unsafe environment.
5) Looking at the fault tree gives no idea on how the software behavior affects or is affected by the faults and failures.

Through the process of failure integration to the gas burner, we got a better understanding about how the components of both software and hardware affect, and are affected by, failures. The process identifies the safety critical software components that are responsible for the failures. In other words, the integration process identifies that the air valve and gas valve from the gas burner are safety critical components and should be given special attention through the system development process. Through this integrated model, failure becomes part of the software behavior and not a separate entity that targets only the hardware components. The integrated model in Fig. 28 shows that the *excess of gas* fault can be easily identified and understood through its interaction with the system functionalities. In addition, the correct behavior of the system is, to a great extent, separated from the faulty behavior. Only minor modifications

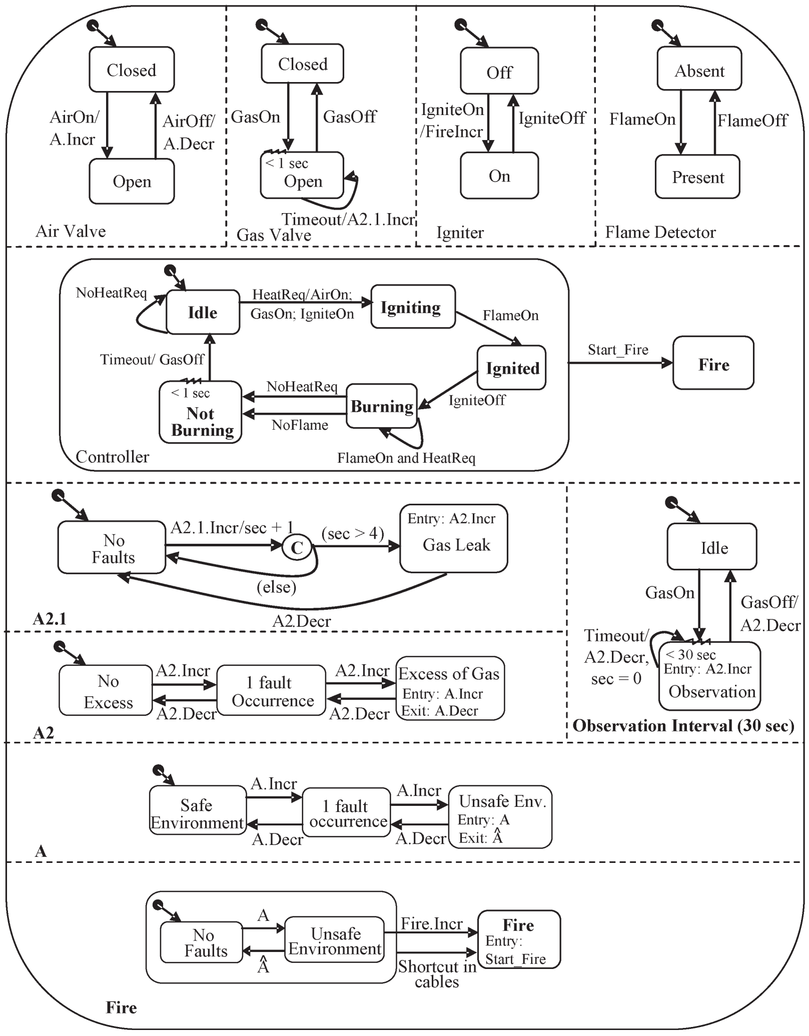


Fig. 28. Modified statechart for the gas burner.

were done to the functional behavior. The newly added fault tree components can be easily spotted from the statechart. This improves the understanding of separated interest, such as focusing only on the functional or faulty behavior of the gas burner.

C. Validation

The specification of a computer-controlled system describes the behavior, functionality, and restrictions of the system. The role of the specification is to guide the implementation process so that the system meets its requirements. In other words, if the system's behavior and functionality follow precisely the system's specifications, then the requirements should not be violated. Therefore, it is crucial for the specifications to be correct, complete, and consistent so that the system requirements are met successfully. Validation is a fundamental step in proving that the specifications have no ambiguities, errors, conflicts, or inconsistencies. Popular requirement validation techniques, such as the formal technical review, only consider the correct behavior of the system during the validation process. The aim of our method is to demonstrate that integrating failure in the behavior of the system plays an important role in requirement validation.

In the case of the gas burner, the *fire* failure deals with the gas burner specification that is responsible for realizing the requirement that "at all times, the gas concentration in the surroundings of the gas burner should not exceed a certain threshold." The integration process identifies the system functionalities and behavior that are in concern for this specific failure. Looking at the integrated model, particularly the *Fire* component, shows that, for the fire to occur, two conditions should take place in a certain order (keeping in mind that the *shortcut in cables* event is only depicted in the statechart representation for demonstration purposes). Therefore, the gas burner will be on fire with the occurrence of the *A* event followed by the *Fire.Incr* event. Through the IFSS model, it is easy to identify the components or states that are responsible for triggering these two events (the conditions that cause the fire failure). The integrated model shows that a transition to the Igniter *On* state will trigger a *Fire.Incr* action, while a transition to an *Unsafe Env* state in the A component will trigger the *A* action. For the *Unsafe Env* to occur, as can be seen in the A component, a transition to the Air Valve *Open* state and a transition to the *Excess of Gas* state should occur at the same time. In order for the *excess of gas* to occur, as indicated by the A2 component, two *A2.Incr* events should be present at the same time. The integrated model shows the inexistence of any restrictions by the system specification in order to prevent the two conditions from the *Fire* component from occurring simultaneously. Therefore, the integration process showed that an implemented gas burner that follows these specifications has a safety vulnerability of a fire occurrence. In other words, the gas burner statechart with respect to the gas concentration requirement is not complete and violates this requirement when an ignition is present while the burner environment is unsafe (the gas concentration exceeds 4 s). The current specification is also inconsistent because it asks the gas burner to perform something (an ignition attempt with an excess concentration of gas) that is impossible to do, and as a result, the fire starts. The integration process of the fire occurrence failure into the gas burner statechart was capable of showing that the specifications with respect to the requirements are inconsistent, incomplete, and thus incorrect.

The IFSS model also helps in identifying and representing system constraints in the integrated model in order to correct the specification and thus prevent the fire occurrence failure. In the original gas burner statechart, without the failure integration, this problem can be partially solved by adding a constraint (a time-out of 4 s) to the *Igniting* state in the Controller component. This restriction does not make the specification completely correct because fire can occur in a repeated attempt (for instance, 4 s caused by the time-out followed immediately by another attempt with a delay in ignition of one or more seconds) to ignite the gas burner. A complete solution to this problem requires the statechart to have the representation of the observation interval of 30 s in order to add another restriction. This restriction cannot be applied to the original statechart, while it can be applied to the integrated model. Therefore, the integrated model acts as a basis model for the prevention of failures and thus acts as a safety model.

D. Summary

The inclusion of failure into the system specification improves the software quality. Safety can be stressed, and weaknesses in the design can be mitigated at an early stage in the development process. The case study showed that the IFSS model helps in depicting how the system behaves under failure and in the identification of safety vulnerabilities. Other advantages and findings are discussed in the following paragraphs.

First, not all failures have the same impact on the validation process. Some failures help in the identification of inconsistencies, incompleteness, or missing constraints in the system specifications. This was the case with the gas burner case study. The *Fire* failure showed that the gas burner specification was incorrect. The failure also identified the required restrictions which are to be added to the specifications. On the other hand, other failures only prove that the requirements are met by the specification. In that case, the integration of failures confirms that the functional behavior correctly implements the system requirements.

Second, only minor modifications were applied to the functional behavior. This can be seen in the gas burner case study. Only action events and a failure state were added to the functional components. These action events and all of the action events in the failure components do not trigger or affect the functional behavior. Therefore, the functional behavior is not affected or altered during the integration process.

Third, our approach integrates one fault tree at a time. This choice of integration is beneficial because it can detect potential conflicts and inconsistencies between fault trees. In certain cases, the order in which fault trees are integrated might have different or incorrect IFSS models. This is the case only when two or more failures cause a conflict to the system behavior. The IFSS model detects these conflicts and prevents inconsistencies in the system behavior.

V. Related Work

The necessity to avoid failures in a safety critical system is crucial in order to ensure that system services can be trusted and thus can be considered to be safe. Although organizational practices that target safety are useful for enhancing it, they are not that effective, and they face difficulties when applied to complex large scale safety critical systems [21]. Therefore, a more rigorous approach is needed for the analysis, identification, and correction of hazards. The early consideration of faults with software was mainly in the area of fault-based testing [5], [22]. This paper focuses mainly on syntactic errors and neglects semantic errors which are harder or more complex to identify, analyze, and correct. In [23], FTA is applied for the first time in software safety analysis, while in [24], different approaches are discussed and compared. In [25], the process of constructing a system safety model from the physical model is described.

Safety Requirements: The fault trees in [26] are used to deduce and identify the requirements in an intrusion detection system, and the notion of minimal cut sets is used in the analysis. A cut set stands for the set of system components that trigger a system failure if they simultaneously fail, while a minimal cut set is the minimum subset of components that will cause the system failure when they fail altogether; otherwise, the failure will not occur [27]. In [6], software requirements from fault trees are interpreted by using a common semantic model for both the safety analysis and software requirement specifications. This common model is constructed through the use of duration calculus [28], which is a real-time interval logic. Through this common model, the deduction of safety requirements is done by considering every fault tree on its own and then making sure that the root node (duration calculus formula) does not occur. Derivation steps for every gate (AND, OR, and priority AND) are given because the requirement deduction from the duration calculus model differs from one gate to another. On the other hand, in [29], the authors try to solve the decomposition problem and guarantee the correctness of the safety model through the transformation of a fault tree into observational transition systems (OTSs). Then, safety requirement specifications are directly deduced from OTSs through the use of CafeOBJ (a formal specification language).

The focus of our approach is not only on the identification and deduction of safety requirements. Through the integration of fault trees into the statechart, the resultant IFSS model depicts both the functional and faulty behavior of a system.

This IFSS model not only represents the safety requirements but also allows the validation of both the functional and safety requirements. In addition, the IFSS model helps in the identification and representation of system constraints in the integrated model in order to mitigate failures or correct the functional and safety specifications.

Integrating Safety Analysis With Functional Specifications: In [30], the authors point out that traditional safety analysis techniques, such as FTA, are constructed through the use of the informal description of the system and thus might result in some incorrectness, incompleteness, and ambiguities. To solve this problem, they propose to apply FTA to formal methods, particularly statecharts. Their experiment was only demonstrated through an example, where system statecharts were used to force correctness and completeness on the events and subevents of a fault tree in order to define the formal semantics of fault trees using duration calculus. Their suggestions were that the specification model should be built separately from the construction of the fault trees but each model's construction process should be influenced by the other process.

In [31], an event sequence graph that represents both the system and its environment is described. This model is capable of representing certain selected risk patterns through the use of regular expressions. Through the ordering of risk levels, various mitigation scenarios are addressed via the use of a defense matrix. From these regular expressions, tests that are intended for safety requirements can be derived. In [32], the Formale Methoden und Sicherheitsanalyse (ForMoSa) approach is introduced to integrate failures into the correct system behavior to formally analyze critical systems. The approach separates the occurrence pattern of failure from the representation of failure in the system model [33]. The authors use three rules to integrate failures into the system statechart. In comparison to our work, the authors do not mention or investigate the problem of how to transform the results of the FTA into statechart notations. In other words, they do not model the failure itself with the faults and system components that are responsible for its occurrence. Therefore, they do not identify the software components that are responsible for the failure and how to correct the model. They only consider the persistence pattern of the failure and the effect of the failure on the system. In addition, the resultant model is greatly modified, thus making it difficult to identify the correct behavior from the faulty one.

The authors of [5], [7], and [8] propose a fault-based approach by integrating fault trees with the statechart model in order to generate test cases. The analysis of fault trees is represented as duration calculus formulas, which is composed of a collection of minimal cut sets. Each minimal cut set formula is then integrated with the statechart system model through the application of a set of conversion rules. Our method differs from the work in [5]–[8], which is done on the integration of fault trees into statecharts, and can be seen from three aspects. The first difference is in the representation of the semantics of fault trees. Their approach represents the analysis of fault trees in a separate model that is based on duration calculus. The usage of an intermediary model introduces time overhead, might restrict or modify the initial meaning of the representation, and can make the automation process (if possible) harder. Our method uses conversion steps that directly transform every gate and its inputs into statechart notations. A Boolean expression formula that represents the analysis results is only used as guidance for the conversion steps and can be easily dropped and replaced by a fault tree that shows the results of the analysis.

The second difference is in the way that the integration to the statechart is done. The authors of [5], [7], and [8] base their integration on minimal cut sets, where each minimal cut set is integrated one at a time into the statechart. Usually, cut sets, system functions that result in a hazard when combined together, concentrate more on the system and its components rather than on the software and its specification through statecharts. Therefore, fault tree integration based on minimal cut sets diverts from the focus of the specifications of software systems. Our method tries to keep the specification simple and understandable through a well-structured modular representation of each failure. In addition, the assignment of a meaning in [5], [7], and [8] (a duration calculus formula) for the leaf nodes in the cut set is done, and is not shown how, before the conversion rules are applied. Determining whether a leaf node is an event, a state, a state transition, or a bounded state is not a trivial step. This process (transformation into duration calculus formulas) should check with the system statechart in order to identify whether this leaf node can be represented by the system or should be considered as a safety assumption. Therefore, it is better for this step to be done during the integration with the statechart, where a leaf node can be easily identified whether it can be represented by the system or not. Our method takes this issue into consideration and directly integrates each gate one at a time with its inputs to the statechart notations without first assigning formulas for each leaf node. The last difference between our approach and those in [5], [7], and [8] is that their method does not consider notations such as time or counters. In other words, the conversion rules in [5], [7], and [8] for the leaf nodes that are bounded in time or that require counters do not have equivalent statechart representations.

Model-Driven Safety Analysis Techniques: Model-driven safety analysis techniques emerged as a response to the limitations found in traditional techniques such as FTA and failure modes and effects analysis. In [34], Lisagor *et al.* give an overview of two of the prominent model-driven safety analysis approaches: failure logic modeling and failure injection. In failure logic modeling, the approach is concerned with the representation of the failure behavior of a system in a hierarchical component-based modular approach. On the other hand, failure injection, such as that in [35], introduces failures into formal models and identifies, through the use of model checking, the behavior that is hazardous or unsafe to the system. Our approach is not a model-driven safety analysis technique, where, in these models (particularly failure logic ones), the analysis is done through the generation of traditional safety analysis techniques (such as fault trees) or other probabilistic models (such as Petri nets). On the contrary, our approach is concerned with the integration of fault trees with the functional specifications of a system and with the benefits that can be achieved from this type of integration throughout the system development process. Therefore, our main focus is to have the safety analysis be part of the system design as one model, rather than dealing with separate models. In addition, model-driven safety analysis techniques are applied to the system architecture [36], while our approach concentrates on functional behaviors.

In [37] and [38], the authors propose a component-based safety analysis model, the state event fault tree (SEFT), that improves on the semantic weaknesses of fault trees when dealing with software. The model differentiates between states and events and thus narrows the semantic difference between safety analysis and the system model. The SEFT model allows quantitative analysis through the transformation of SEFT components into deterministic and stochastic Petri nets (DSPN). The resultant Petri net model is not intended, due to its complexity, to be used and analyzed by software engineers or safety analysts. It is neither intended to emphasize safety through the software development life cycle. The model is only generated in order to perform quantitative analysis. The focus is on whether the system model passes as being safe; therefore, the result of the analysis (or the DSPN model) does not show the components that are responsible for the failure and how to correct the model.

The Architecture Analysis and Description Language's (AADL) error model annex [39] is another model-driven safety analysis technique that differentiates between states and events. The representation of failure is closer to the system behavior. The AADL [40] is a modeling language for the description and analysis of the architecture of a system. The error annex is used to supplement the architecture description done by the AADL with safety- and reliability-related information. The AADL's error annex allows either qualitative safety analysis through the generation of standard fault trees or quantitative analysis through the transformation of the error model into generalized stochastic Petri nets (GSPNs). In [36], the SEFT and the AADL are compared to other model-driven safety analysis techniques.

The analysis of both the SEFT and AADL's error annex does not take into consideration the mismatches between the fault analysis and the system model, where some events cannot be represented by the system behavior. In addition, the analysis does not consider composite definition or the "decomposition problem" [29], [30]. Our approach focuses on qualitative analysis rather than quantitative analysis because software failures are deterministic in nature [41]. As the focus of the IFSS model is on the software faults and failures and their effect on both the software and the whole system, the concentration will be on the reasons for failure occurrences rather than the probability of these occurrences (which can only be applied to hardware components).

VI. Conclusion

We have introduced a new approach that integrates fault trees into statecharts. An IFSS model obtained from the integration shows how the system behaves when a failure condition occurs and acts as the basis model to ensure safety through requirement validation. The IFSS model eliminates difficulties that are encountered through the use of separate models such as ambiguities and the faults' order of occurrence.

The motivation for this paper was due to the importance of ensuring safety and improving on the limitations and ambiguities of the previously proposed methods. Our goal was to maintain the semantics of both models in order for them to be integrated. The case study was chosen to demonstrate different aspects of fault tree notations and how they can be integrated into system statecharts. The example showed the successful application of the conversion rules and transformation steps which transformed hazards into statechart notations.

The scalability of our approach largely depends on the scalability of the fault trees and statecharts. Fault trees have been applied successfully to safety critical systems with a wide range of complexity. Some complex systems, such as the space shuttle, can have hundreds of fault trees [11], and a single fault tree can have hundreds of gates and events [42]. In our approach, a complex tree can be handled by first dividing the tree into independent modules [42] and then working on each module separately as if they were different fault trees. Since an IFSS model represents the low level behavior of a failure, integrating complex fault trees will increase the number of states and transitions in the resultant IFSS model. However, statecharts can deal with a large number of states through the use of hierarchical decomposition and concurrence.

This paper integrates two models that are heterogeneous in both structure and semantics. The presence of a safety analyst might still be required in the integration process to clarify the mismatches and ambiguities. Our methodology makes it possible to include safety into the software design process at an early stage in order to facilitate the automation process. This allows the safety team to collaborate with the software team while developing the behavior of the system (and developing the IFSS model). This collaboration is manifested through the selection of the desired fault trees which are to be integrated and in resolving semantic and syntactic mismatches between the two models. Future work will concentrate on managing and assisting this type of collaboration between the safety analysts and the system development team through a tool support for the IFSS model. This tool will be the starting point for the proposed approach to be automated.

Another concern is the correctness of the conversion steps, where the conversion rules are intended to preserve the meaning of the fault tree gates when transformed into statechart representations. Through the informal explanation and evaluation of the conversion rules, these statechart representations were demonstrated to have an identical meaning to the gate semantics. In addition, the case study demonstrated that the successful application of the conversion rules and transformation steps resulted in a correct statechart representation for the failure. Nevertheless, it is desirable to formally prove that the conversion rules preserve the meaning of the gates and correctly represent the fault tree behavior. We plan to use model checking as a formal method for verifying the correctness of IFSS models.

References

[1] J. Rushby, "Critical system properties: Survey and taxonomy," *Reliab. Eng. Syst. Saf.*, vol. 43, no. 2, pp. 189–219, 1994.

[2] *Fault Tree Handbook*, U.S. Nucl. Regulatory Comm., Washington, DC, 1981, NUREG-0492.

[3] D. Harel, "Statecharts in the making: A personal account," in *Proc. 3rd ACM SIGPLAN Hist. Program. Lang. Conf.*, 2007, pp. 5–43.

[4] O. El-Ariss, D. Xu, W. E. Wong, Y. Chen, and Y. Lee, "A systematic approach for integrating fault trees into system statecharts," in *Proc. 32nd Annu. IEEE Int. COMPSAC*, 2008, pp. 120–123.

[5] M. A. Sanchez and M. A. Felder, "A systematic approach to generate test cases based on faults," in *Proc. ASSE*, 2003.
[6] K. M. Hansen, A. P. Ravn, and V. Stavridou, "From safety analysis to software requirements," *IEEE Trans. Softw. Eng.*, vol. 24, no. 7, pp. 573–584, Jul. 1998.
[7] M. Sanchez, J. Augusto, and M. Felder, "Fault-based testing of E-commerce applications," in *Proc. 2nd Workshop Verification Validation Enterprise Inf. Syst.*, 2004, pp. 66–71.
[8] A. Dasso and A. Funes, *Verification, Validation and Testing in Software Engineering*. Hershey, PA: Idea Group Publ., 2007.
[9] M. Walker and Y. Papadopoulos, "Qualitative temporal analysis: Towards a full implementation of the fault," *Control Eng. Pract.*, vol. 17, no. 10, pp. 1115–1125, Oct. 2009.
[10] B. Kaiser, P. Liggesmeyer, and O. Mäckel, "A new component concept for fault trees," in *Proc. 8th Australian Workshop Saf. Crit. Syst. Softw. (SCS)*, 2003, pp. 37–46.
[11] W. Vesely, J. Dugan, J. Fragola, J. Minarick, and J. Railsback, *Fault Tree Handbook With Aerospace Applications*. Washington, DC: NASA, 2002.
[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.
[13] F. D. Von Borstel and J. L. Gordillo, "Model-based development of virtual laboratories for robotics over the Internet," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 40, no. 3, pp. 623–634, May 2010.
[14] D. Harel and M. Politi, *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*. New York: McGraw-Hill, 1998.
[15] D. Drusinsky, *Modeling and Verification Using UML Statecharts*. Amsterdam, The Netherlands: Elsevier, 2006.
[16] M. von der Beeck, "A comparison of statecharts variants," in *Proc. Formal Techn. Real Time Fault Tolerant Syst. (LNCS)*, 1994, pp. 128–148.
[17] G. Lüttgen, M. von der Beeck, and R. Cleaveland, "A compositional approach to statecharts semantics," in *Proc. 8th ACM SIGSOFT FSE*, New York, 2000, pp. 120–129.
[18] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Trans. Softw. Eng. Method*, vol. 5, no. 4, pp. 293–333, Oct. 1996.
[19] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992.
[20] K. M. Hansen, A. P. Ravn, and H. Rischel, "Specifying and verifying requirements of real-time systems," in *Proc. Conf. Softw. Citical Syst. (SIGSOFT)*, 1991, pp. 44–54.
[21] M. Grabowski, Z. You, H. Song, H. Wang, and J. R. W. Merrick, "Sailing on friday: Developing the link between safety culture and performance in safety-critical systems," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 40, no. 2, pp. 263–284, Mar. 2010.
[22] T. Y. Chen, D. D. Grant, M. F. Lau, S. P. Ng, and V. R. Vasa, "BEAT: A Web-based Boolean expression fault-based test case generation tool," *Int. J. Distance Edu. Technol.*, vol. 4, no. 2, pp. 44–56, 2006.
[23] N. G. Leveson and P. R. Harvey, "Analyzing software safety," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 5, pp. 569–579, Sep. 1983.
[24] N. G. Leveson, "Software Safety: Why, what and how," *ACM Comput. Surv.*, vol. 18, no. 2, pp. 125–163, Jun. 1986.
[25] S. Liu and J. A. McDermid, "A model-oriented approach to safety analysis using fault trees and a support system," *J. Syst. Softw.*, vol. 35, no. 2, pp. 151–164, Nov. 1996.
[26] G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, and R. Lutz, "A software fault tree approach to requirements analysis of an intrusion detection system," *Requirements Eng.*, vol. 7, no. 4, pp. 207–220, Dec. 2002.
[27] R. Billinton and R. N. Allan, *Reliability Evaluation of Engineering Systems*. New York: Plenum, 1983.
[28] C. Zhou, C. Hoare, and A. P. Ravn, "A calculus of durations," *Inf. Process. Lett.*, vol. 40, no. 5, pp. 269–276, Dec. 1991.
[29] J. Xiang, K. Ogata, W. Kong, and K. Futatsugi, "From fault tree analysis to formal system specification and verification with OTS/CafeOBJ," *Comput. Softw., JSSST*, vol. 23, no. 3, pp. 134–146, 2006.
[30] W. Reif, G. Schellhorn, and A. Thums, "Safety analysis of a radio-based crossing control system using formal methods," in *Proc. 9th IFAC Symp. Control Transp. Syst.*, 2000, pp. 289–294.
[31] F. Belli, A. Hollmann, and N. Nissanke, "Modeling, analysis and testing of safety issues—An event-based approach and case study," in *Proc. 26th Int. Conf., SAFECOMP*, 2007, pp. 276–282.
[32] F. Ortmeier and W. Reif, "Formal safety analysis of transportation control systems," in *Proc. TRAIN@SEFM Workshop*, 2005.
[33] F. Ortmeier, W. Reif, and G. Schellhorn, "Formal safety analysis of a radio-based railroad crossing using deductive cause–consequence analysis," in *Proc. 5th EDCC*, 2005, pp. 210–224.
[34] O. Lisagor, J. A. McDermid, and D. J. Pumfrey, "Towards a practicable process for automated safety analysis," in *Proc. 24th Int. Syst. Saf. Conf.*, 2006, pp. 596–607.
[35] A. I. McInnes, B. K. Eames, and R. Grover, "Formalizing functional flow block diagrams using process algebra and metamodels," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 41, no. 1, pp. 34–49, Jan. 2011.
[36] L. Grunske and J. Han, "A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models," in *Proc. 11th IEEE High Assur. Syst. Eng. Symp.*, 2008, pp. 283–292.
[37] B. Kaiser, C. Gramlich, and M. Förster, "State-event-fault-trees—A safety analysis model for software controlled systems," *Reliab. Eng. Syst. Saf.*, vol. 92, no. 11, pp. 1521–1537, Nov. 2007.
[38] L. Grunske, B. Kaiser, and Y. Papadopoulos, "Model-driven safety evaluation with state-event-based component failure annotations," in *Proc. 8th Int. Symp. CBSE*, 2005, pp. 33–48.
[39] P. H. Feiler and A. E. Rugina, *Dependability Modeling With the Architecture Analysis and Design Language (AADL)*. Pittsburgh, PA: Carnegie Mellon Univ., 2007.
[40] P. H. Feiler, D. P. Gluch, and J. J. Hudak, *The Architecture Analysis and Design Language (AADL): An Introduction*. Pittsburgh, PA: Carnegie Mellon Univ., 2006.
[41] P. Fenelon and J. A. McDermid, "An integrated toolset for software safety analysis," *J. Syst. Softw.*, vol. 21, no. 3, pp. 279–290, Jun. 1993.
[42] Y. Dutuit and A. Rauzy, "A linear time algorithm to find modules of fault trees," *IEEE Trans. Rel.*, vol. 45, no. 3, pp. 422–425, Sep. 1993.

Omar El Ariss received the B.S. and M.S. degrees in computer science from Lebanese American University, Beirut, Lebanon, in 2001 and 2005, respectively. He is currently working toward the Ph.D. degree in the Department of Computer Science, North Dakota State University, Fargo.

His research interests include software safety, software security, software modeling and verification, and software testing.

Dianxiang Xu (SM'01) received the Ph.D. degree in computer science from Nanjing University, Nanjing, China.

From 2003 to 2009, he was an Assistant Professor of computer science with North Dakota State University, Fargo. He is currently an Associate Professor with the National Center for the Protection of the Financial Infrastructure, Dakota State University, Madison, SD. His research interests include software security and safety, software testing, applied formal methods, and computer forensics.

W. Eric Wong (SM'00) received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN.

He was with Telcordia (formerly Bellcore) as a Project Manager for Dependable Telecom Software Development. He is currently an Associate Professor in computer science with the University of Texas at Dallas (UTD), Richardson.

Dr. Wong is the Secretary of the Association for Computing Machinery Special Interest Group on Applied Computing and is a member of the Administrative Committee of the IEEE Reliability Society. He was the recipient of the Quality Assurance Special Achievement Award from Johnson Space Center, National Aeronautics and Space Administration (1997).