The Introduction of a Safety-based State Machine for Improved System Safety Analysis

Article · August 2016

1 author:

Samuel Benton
University of Texas at Dallas
10 PUBLICATIONS **195** CITATIONS

SEE PROFILE

The Introduction of a Safety-based State Machine for Improved System Safety Analysis

Samuel Benton
The University of Texas at Dallas

Abstract

Large or complex systems are often developed with models that describe system functional specifications and safety specifications independently. Developing systems with these multiple independent models can unknowingly introduce safety hazards into the system. This research introduces and describes a model, called the Safety State Machine, which integrates functional specifications, as described from UML state machine diagrams, and safety specifications, derived from Fault Tree diagrams, under one model. The Safety State Machine is a UML state machine which uses the principle of UML orthogonality to model systems and subsystems rather than individual objects. By incorporating all entities which affect system functionality, the status of system safety hazards can be measured, tracked and preemptively resolved. Should a fault still occur within the system, the Safety State Machine diagram provides provisions for fault damage mitigation and fault resolution actions. Furthermore, the Safety State Machine provides a framework through which system safety can be tested. Additional models and heuristics which assist in the complete development of the Safety State Machine are provided and described. These tools are demonstrated in an example case study and ultimately enhance the system's safety while simultaneously developing a more complete understanding of the system as a whole. From the Safety State Machine and supporting models, system safety can be developed and enhanced in a more complete manner which can consequently decrease the frequency and severity of safety hazards in systems.

Section 1 - Introduction

For all but the smallest and simplest of systems, a given system uses many models to describe various aspects of the system. A set of the more important models are those that describe the system's functional specifications. These models describe ways in which the system should behave or operate under expected conditions. The Unified Modeling Language (UML) contains a set of diagrams called UML Behavioral Diagrams which describe functional specifications in various contexts. UML communication diagrams, for instance, describe communication between objects in a system while UML state machine diagrams describe the internal design of individual objects [1]. Fault analysis models, such as Fault Tree Analysis (FTA), are models which describe ways in which the system should not behave or operate,

typically in specific unideal conditions [2, 3]. Safety engineers can then derive safety specifications from these fault analysis models.

Systems developed without any model integrating functional specifications and safety specifications are more likely to possess safety oversights [4, 5, 6]. Most functional behavioral models have limited capabilities to represent safety specifications or hazards in a system. Similarly, diagrams representing safety specifications have little, if any, capability to model functional specifications [4]. A model integrating functional and safety specifications may reduce the severity of existing hazards, highlight undiscovered hazards, and provide a more comprehensive overview of the system [7]. Section 2 introduces and describes this integrated model alongside additional supporting models and techniques. Section 3 uses the aforementioned items in a case study. Sections 4 concludes this paper and describes possible future research for the aforementioned items.

Section 2 - Research Diagrams and Heuristics

The State Hierarchy Tree

Overall system safety is reflective of the diagrams used to model safety and the processes of building the system to reflect the diagrams. Inaccurate safety models result in the system's inability to correctly handle a fault. Likewise, incomplete safety models result in the lack of system structures or objects to mitigate or otherwise faults [8]. As such, a model or process which assists in comprehensive fault generation makes fault analysis models and the resultant safety specifications more complete.

The State Hierarchy Tree is a model which can highlight undetected hazards in system design and can provide insight into new causes which contribute to existing hazards. A State Hierarchy Tree lists the states of an object's state machine in a hierarchical manner. An example of an object's state machine and the derived State Hierarchy Tree are shown in Figure 1 and Figure 2. The formal steps for generating a State Hierarchy Tree from a state machine are as follows:

1. Set the top level node's name to the state machine name
2. For every state or composite state **s** in state machine **m**
    a. If **s** is not a substate, then add **s** as a child to the top level node
    b. If **s** is a substate of composite state **c**, then add **s** as a child to **c**
    c. If **s** contains regions $\mathbf{r_0}$ **-** $\mathbf{r_n}$, then add $\mathbf{r_0}$ **-** $\mathbf{r_n}$ as children to **s**
    d. If **s** is enclosed in region **r**, then add **s** as a child to **r**
3. For every region **r** in state machine **m**
    a. If **r** is not in enclosed in any composite state, then add **r** as a child to the top level node
    b. If **r** is enclosed in composite state **c**, then add **r** as a child to **c**
    c. If **r** contains states $\mathbf{s_0}$ **-** $\mathbf{s_n}$, then add $\mathbf{s_0}$ **-** $\mathbf{s_n}$ as children to **r**

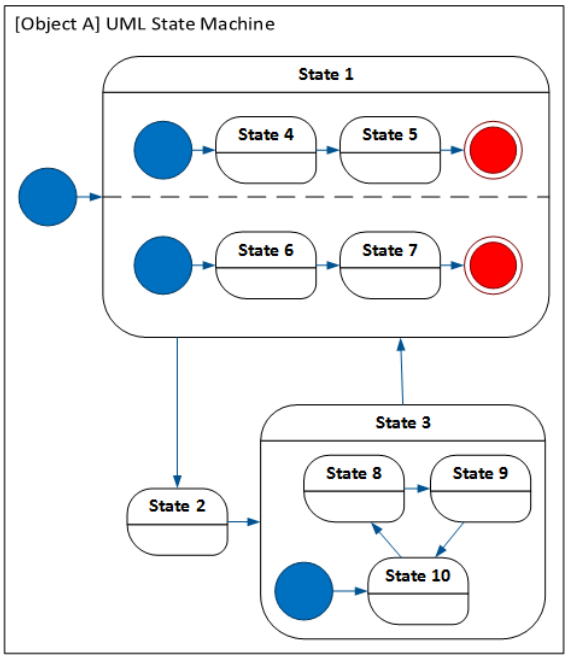

Figure 1: An example UML state machine diagram, with a corresponding State Hierarchy Tree diagram shown in Figure 2.

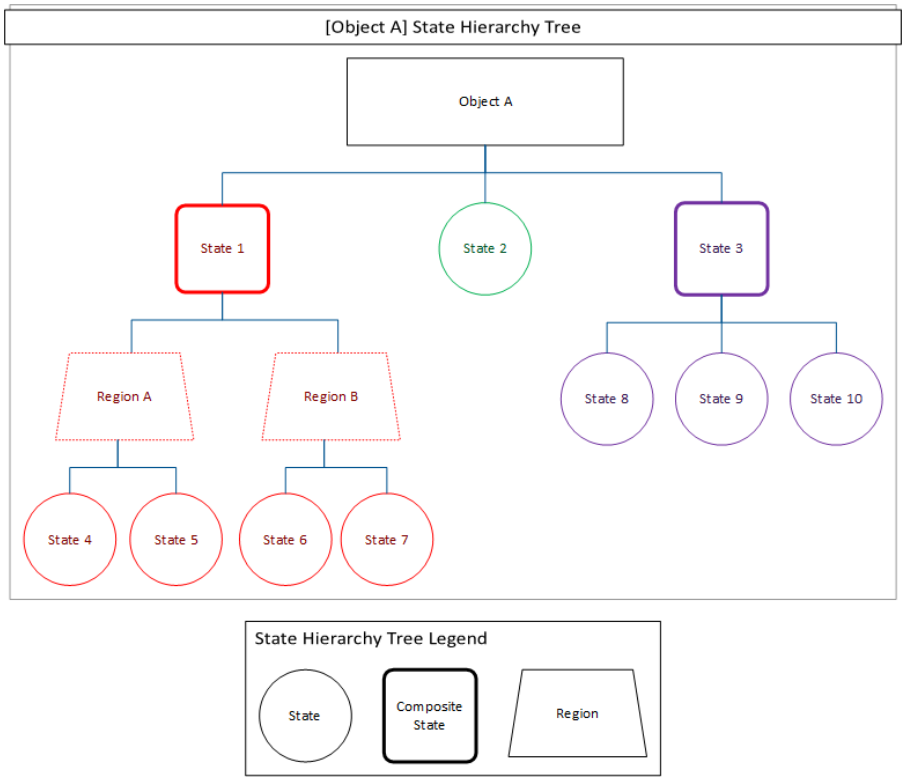

Figure 2: An example of a State Hierarchy Tree (top) and the corresponding legend (bottom) for the tree. State Hierarchy Trees can represent orthogonality, similar to UML state machines, as shown in the subtree of State 1

Often, unexpected faults result from the incomplete analysis of object interaction within a system. With respect to UML state machines, certain combination of states between multiple objects may be hazardous and may result in faults if not properly handled. The aggregation of multiple State Hierarchy Trees into a truth table form provides a framework for various combination of states to be analyzed in an inductive bottom-up manner for their hazard potential. This technique assists with the discovery of undetected hazards and hazard causes in system design. The steps for creating this State Hierarchy Tree Truth Table from a set of State Hierarchy Trees are listed as follows:

1. For every State Hierarchy Tree that will be analyzed
    - add every leaf node state as a column, called a state column, to the table
    - optionally, add every composite state as a column to the table
2. Add a 'Potential Hazard' column to the table
3. Add a 'Hazard Details' column the table

The state columns represent whether the given states are active (1) or inactive (0). Unless the given state hierarchy tree contains orthogonal composite states, only one state per tree may be active, reducing the combinatorial explosion seen from traditional Boolean truth tables. Boolean algebra simplification techniques can be applied to a State Hierarchy Tree Truth Table to detect patterns and reduce the size of the table. When the simplification process is completed, subjective analysis can be performed to detect underlying patterns and dependencies which contribute to hazards. Don't-care values (x) similar to those used in Karnaugh Maps may be employed in state columns to simplify State Hierarchy Tree Truth.

The Potential Hazard column describes if the row's given combination of active and inactive states results in a potential hazard. The Potential Hazard column may have Yes/No values or some form weighted metric.

The Hazard Notes column provides an area for the description and explanation as to why a given combination of states does or does not result in a hazard. An example of a State Hierarchy Tree Truth Table analyzing two objects is shown in Table 1.

Row Number	Object 1			Object 2			Potential Hazard	Hazard Notes
	s1	s2	s3	s4	s5	s6		
1	**1**	0	0	**1**	0	0	No	
2	**1**	0	0	0	**1**	0	No	
3	**1**	0	0	0	0	**1**	No	
4	0	**1**	0	**1**	0	0	No	
5	0	**1**	0	0	**1**	0	**Yes**	Notes about this hazard
6	0	**1**	0	0	0	**1**	**Yes**	Notes about this hazard
7	0	0	**1**	**1**	0	0	No	
8	0	0	**1**	0	**1**	0	No	
9	0	0	**1**	0	0	**1**	No	

Table 1: Example of using multiple State Hierarchy Trees in a 'truth table' fashion to discover existing hazards in a system. An additional 'Row Number' column is added for easy identification of rows. From this table, it is evident that s2 AND (s5 OR s6) results in a system hazard. From this, Object 1's and Object 2's state machines can be modified to prevent or mitigate the detected hazards.

Fault Tree Conversion Heuristics

The proposed model integrating functional specifications and safety specifications, the Safety State Machine, contains state machines derived from fault trees. Thus, a given fault tree must be compatible with and convertible into state machine notation. This notation requires every intermediate event to be associated with some state in some state machine within the system. Should such a state not exist, then a state which reflects the intermediate event must be inserted into an existing state machine or inserted into a new state machine. Additionally, these intermediate events must be observable, detectable, and measurable These three characteristics ensure that the intermediate event can be represented as a state in a state machine. Basic events are treated as actions, signals, or triggers which cause a state change within an object. These state changes propagate throughout the system causing other state changes in other objects

An example fault tree with incompatible intermediate events is shown in Figure 3. Table 2 possible shows resolution actions for the 'Gas Explodes' intermediate event while Figure 4 shows an example where a state machine was created so the 'Gas Explodes' intermediate event could be associated to a state. Finally, Figure 5 shows a revised version of Figure 3 where all intermediate events are in a state machine compatible notation.

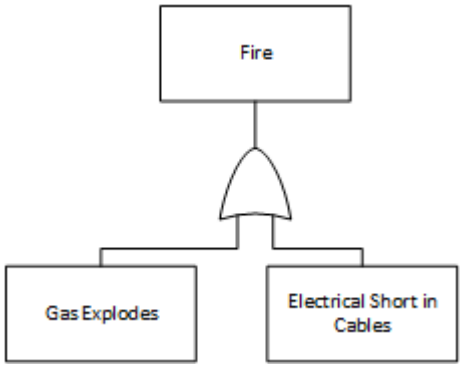

Figure 3: Example fault tree which is incompatible because some of its intermediate events (Gas Explodes) are not observable, detectable, and measurable.

Incompatible Fault Tree Semantics	• Gas Explodes intermediate event ○ **Issue** - This event is not measurable. Explosions are immediate, destructive actions rather than measurable properties or passive actions.

Compatible Fault Tree Semantics	• Active "Dangerous Gas Environment" state
	  ○ **Resolution A** - A "Gas Environment" state machine is created to represent this state. Possible states are a "Safe Gas Environment" and "Dangerous Gas Environment" state. Measure the conditions leading up to the fault and transition to the "Dangerous Gas Environment state when these conditions occur
	  ○ **Reasoning A** – The "Gas Explodes" state is reworked into something that is observable, detectable, and measurable by its own system or some external system

	• Active "Gas Explosion Imminent" state
	  ○ **Resolution B** - Add some set of sensors to the system which accurately predict when a gas explosion is imminent within the system
	  ○ **Reasoning B** - The set of sensors will able to physically detect or accurately infer when a gas explosion is about to occur in the system

Table 2: Example of transforming an intermediate event with incompatible state machine semantics into an intermediate event with compatible state machine semantics.

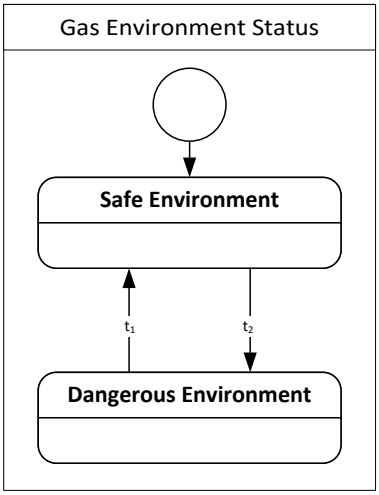

Figure 4: An example where a state machine is created so that an intermediate event (Gas Explodes from Figure 3) can be associated to it. Transitions t_1 and t_2 represent the conditions which makes a dangerous gas environment safe and a safe gas environment dangerous.

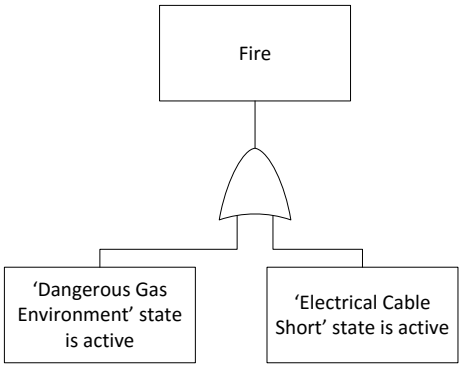

Figure 5: Revised version of Figure 3 with every intermediate event associated to some state.

After a fault tree is in a state machine compatible notation, the fault tree is converted into a set of state machines by applying transformation rules which conserve the tree's Boolean logic semantics. The conversion creates a state machine from a given gate, the gate's inputs, and the gate's output.

For an AND gate, all inputs must occur before the gate triggers [9, 10]. This can be represented in state machine notation by inserting every gate input as a guard condition into one incoming transition to the output intermediate event [9, 11]. The transformation for AND gates is shown in Figure 6.

For an OR gate, the gate triggers if any input occurs. This can be represented in state machine notation by having incoming transitions to the output intermediate event for each input [11, 10]. These transitions contain guard conditions which represents the given input [9]. The transformation for OR gates is shown in Figure 7.

PAND gates are treated as AND gates in the transformation process. To preserve the PAND gate's ordering semantics, the state machines of the inputs are modified such that the transition to the given input transition must occur only when the previous inputs have already occurred [4]. Specifically, for input $\mathbf{i_n}$ and the event's associated state $\mathbf{sr_n}$, the associated state $\mathbf{sr_n}$ must include $\mathbf{sr_{n-1}}$ as a guard

condition to all incoming transitions $\mathbf{t_n}$ of $\mathbf{sr_n}$. Additionally, another similar yet distinct state $\mathbf{sr_m}$ must be included which represents the state when any transition $\mathbf{t_n}$ occurs but guard condition $\mathbf{sr_{n-1}}$ has not occurred.

Inhibit gates are also treated as AND gates with the conditional event acting as a basic event. Voting OR gates and XOR gates can be created through the use of AND gates and OR gates so their specific transformation rules are not further discussed.

In these transformations, the source state are states which can transition into Output State. If no other state transitions into Output State, source state is the inverse of Output state.

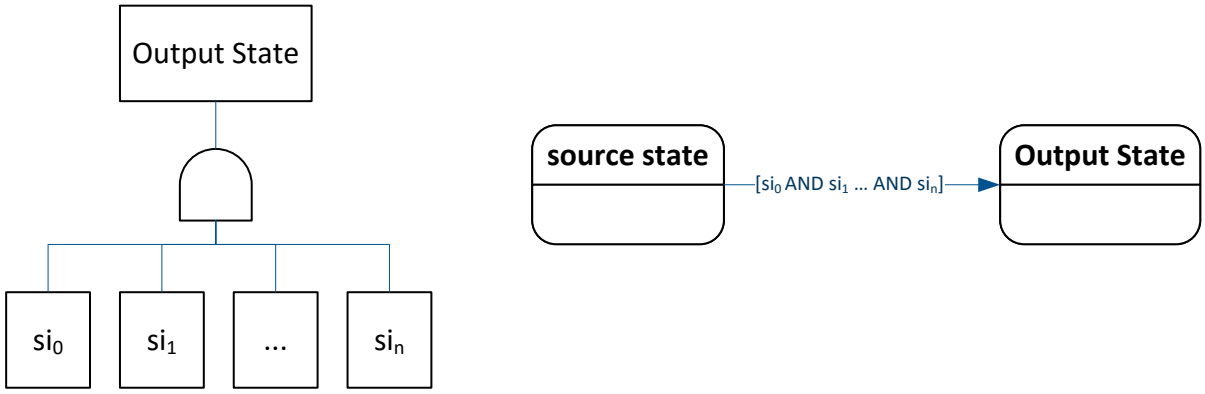

Figure 6: Transformation rule for transforming a fault tree AND gate into a state machine. Each input to the AND gate is used as a guard in the incoming transition to Output State.

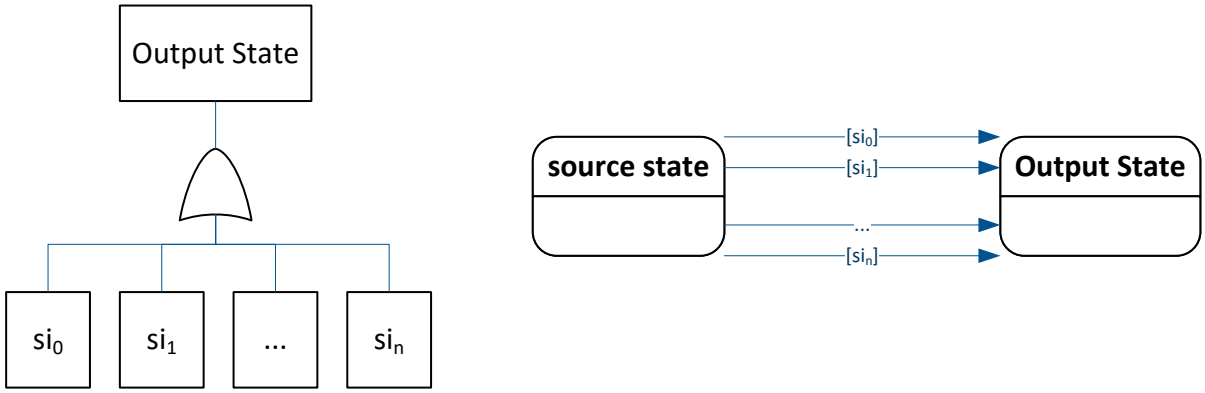

Figure 7: Transformation rule for transforming a fault tree OR gate into a state machine. Each input to the OR gate is a separate transition to Output State.

The Safety State Machine

The Safety State Machine is a UML state machine which uses the principle of state machine orthogonality to model systems and subsystems instead of individual objects. By inserting the system's core object (functional specifications) and the state machines derived from a system's set of fault trees (safety specifications), the Safety State Machine combines and integrates both specification types within one model.

In addition to containing state machines representing the system's core objects and intermediate events from fault trees, the Safety State Machine also contains state machines representing fault tree top-level faults. This allows active observation of the given hazard within the system. Fault tree top-level faults follow the same transformation rules as intermediate events, but the Output State is decomposed into two additional states, shown in Figure 8.

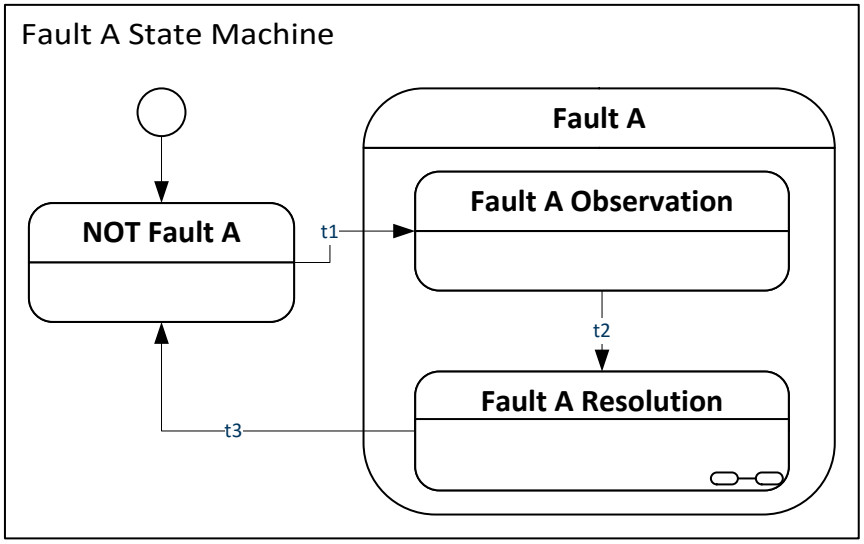

Figure 8: Example transformation of a fault tree's top-level fault into a fault state machine.

The description and explanation of these states are as follows:

- ***NOT FAULT A*** - This state is the initial pseudostate within its orthogonal region. When the fault has not occurred in the system, this state is active. This state has a set of outgoing transitions from the transformation process (grouped into t_1) to the FAULT A OBSERVATION substate within the FAULT A composite state. The set of outgoing transitions are the observable, measurable, invalid combination of states leading to the fault FAULT A. This state has a set of incoming transitions (t_3) from the FAULT A RESOLUTION substate which represent the criteria needed for the system to recover from the given fault, if the fault is recoverable.
- ***FAULT A*** - This composite state is used for simple detection of the representative fault. A fault is considered to have occurred when this state is active. This state may have explicit outgoing transitions to a terminate pseudostate if the fault is unrecoverable or emergency action is needed.
- ***FAULT A OBSERVATION*** - When active, the FAULT A OBSERVATION state represents that FAULT A has occurred but no recovery or resolution actions have initiated. State machine internal activities can trigger to collect data at the time of fault occurrence or to initialize objects needed for fault recovery. This state has a set of incoming transitions (t_1) from NOT FAULT A representing the observable, measurable, invalid combination of states leading to the fault FAULT A. This state has a set of outgoing transitions (t_2) to FAULT A RESOLUTION representing the conditions under which fault resolution, fault recovery, or fault damage reduction actions should initiate.

- ***FAULT A RESOLUTION*** - When active, the FAULT A RESOLUTION state represents that FAULT A has occurred and resolution actions have started. This state has a set of incoming transitions (t_2) from FAULT A OBSERVATION representing the conditions under which fault resolution, fault recovery, or fault damage reduction actions may start after a given fault occurs. This state has a set of outgoing transitions (t_3) from to NOT FAULT A representing criteria needed for the system to recover from the given fault if the fault is recoverable. Internal activities are employable to initiate functions to facilitate fault recovery and log fault timestamps if necessary. This state may have an outgoing transition to a terminate pseudostate if emergency action, such as emergency shutdown, is needed or if the fault is unrecoverable. This state can be a composite state if further detail is necessary for the fault resolution process.

The goal of the of Safety State Machine is to reduce the occurrence of any Fault A state through the design modification of other object's state machines. Thus, incoming transitions to Fault A states, represented by t_1 in Figure 8, are analyzed to determine how to prevent the transition's triggers and guards from occurring. Then, these modifications are implemented in system design which elIn short, design modifications must be made to ensure the Fault A states are active as little as possible.

Using the Models to Enhance System Safety

The collective objective of the aforementioned items is to enhance system safety. The steps to achieve this objective is as follows:

1. Create or otherwise obtain state machines for the system's core objects and system fault trees
2. Convert system fault trees into state machines
3. Aggregate the set of state machines into a Safety State Machine
4. Determine, validate and verify necessary design modifications needed to reduce the occurrence of states representing faults in the Safety State Machine

The proper execution of these steps and development of these models should lead to enhanced system safety and system design without introducing additional errors or hazards into the system.

Section 3 - Case Study

Case Study Introduction

To demonstrate how system safety can be improved through the usage of these models, the models shall be used in a simple case study, described below.

A client has ordered the construction of a smart house which, amongst other things, advertises a novel automatic door system. This automatic door system is operational solely by the smart house system; no human operators can physically open or close the door. The door system has a voice control

panel (hereby referred to as the Voice Control Unit) which signals the door to open when the panel receives input from an authorized voice. After the door has been opened for some threshold of time, 5 seconds, the door system automatically closes the door. Figure 9, Figure 10, and Figure 11 show the state machines for the system's core objects; the Front Door, Voice Control Unit, and External Power Unit. The State Hierarchy Tree for these objects are shown in Figure 12, Figure 13, and Figure 14 respectively.

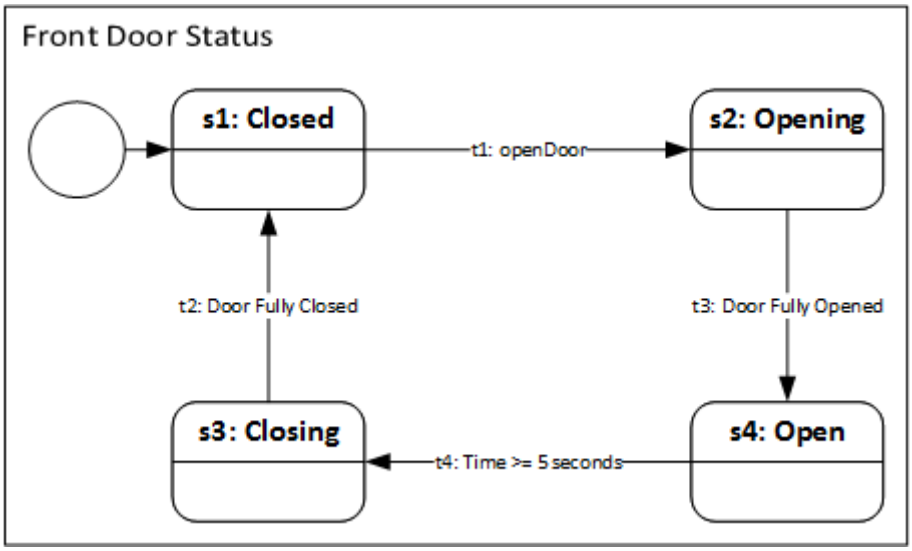

Figure 9: UML state machine diagram representing the 'front door' in the given smart house case study. The door begins to open when it receives an "openDoor" signal from the Voice Control Unit. The door begins closing after the Open state, s4, has been active for 5 seconds.

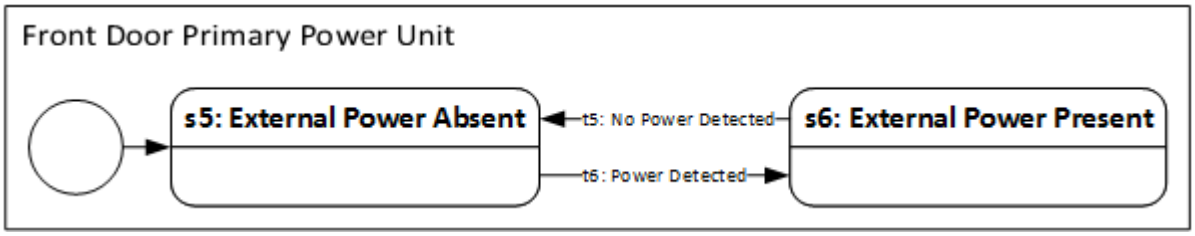

Figure 10: UML state machine diagram representing the status of power for the smart house front door system. State s5 is the default state, thus requiring transition t6 to explicitly trigger for the system to be in a powered state.

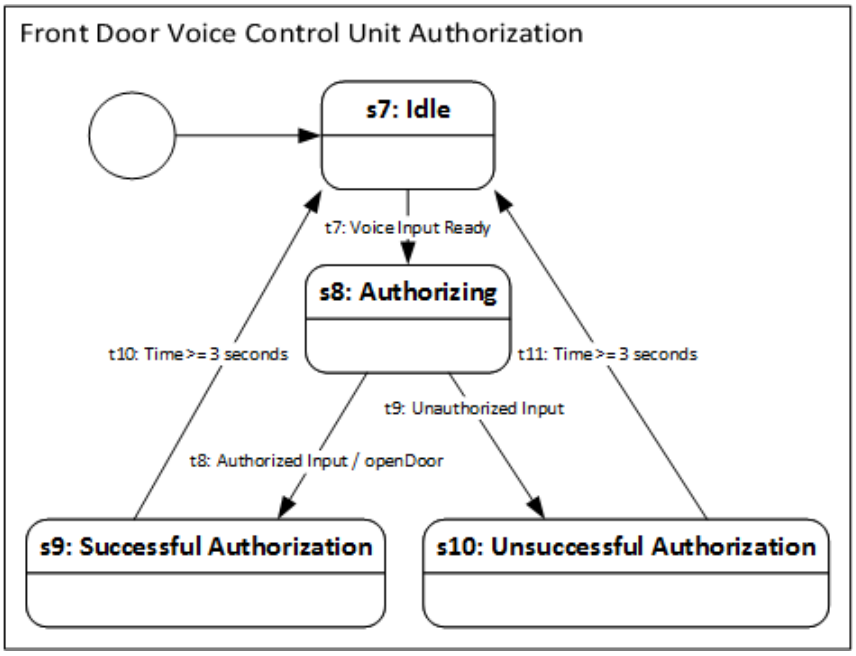

Figure 11: UML state machine diagram representing the Voice Control Unit in the smart house. When the Voice Control Unit receives input, the authorization process begins. Upon successful authorization, an openDoor signal is sent. Three seconds after the authorization returns, successful or unsuccessful, the unit returns to an idle state.

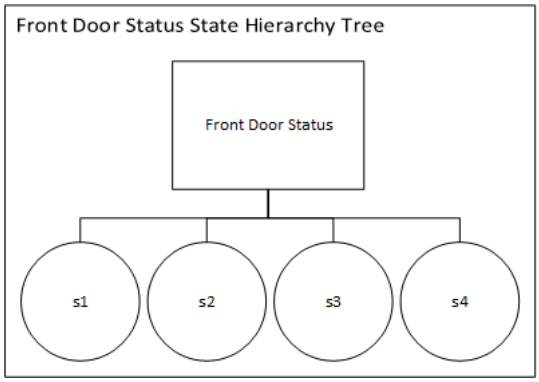

Figure 12: The State Hierarchy Tree derived from the Front Door Status UML state machine.

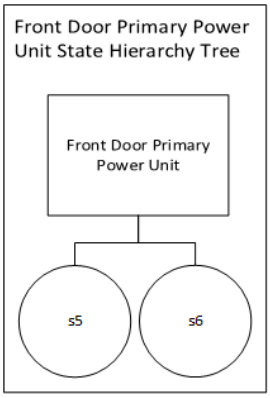

Figure 13: The State Hierarchy Tree derived from the Front Door Primary Power Unit UML state machine.

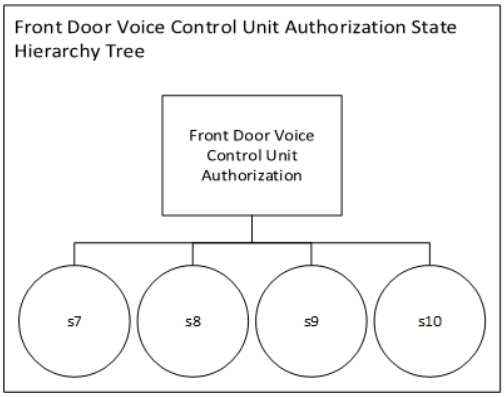

Figure 14: The State Hierarchy Tree derived from the Front Door Voice Control Unit Authorization UML state machine.

Supplied Fault Trees

Since the door automatically closes after five seconds and no human operators can physically move the door, it is possible that some object may get stuck in the door which would be a hazard to the object and the smart house system. Thus, the system's safety team describe this door obstruction hazard in a fault tree shown in Figure 15.

Now that the system's set of state machines and fault trees are supplied, the process to generate the actual Safety State Machine may begin.

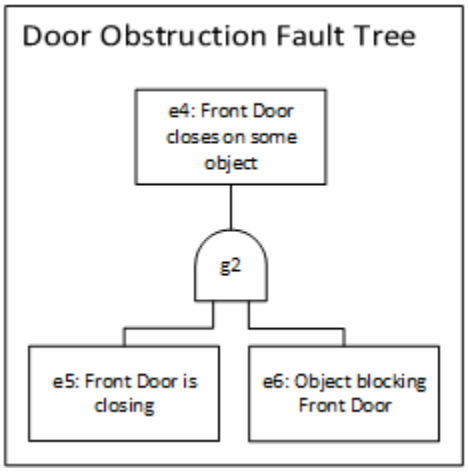

Figure 15: The fault tree diagram representing the Door Obstruction hazard. This fault occurs when the Front Door is closing and some object blocks the Front Door from closing.

Creation of the Safety State Machine

To begin the process, every fault tree must be in state machine compatible notation. As such, Figure 15 must be checked for compatibility which means every intermediate event must be associated with some state in a state machine. The following is the status of this intermediate event association.

- Intermediate Event e5 is already represented by state s3 in the Front Door Status state machine.
- Intermediate Event e6 is not represented by any state currently in the system, so such a state must be created. Since the system also lacks an object for this state, an object much be added to the system. A simple object to add to the system is an obstruction sensor object, hereby referred to as the Door Obstruction Sensor. There are two important states for the Door Obstruction Sensor, a Door Obstructed state (s26) and a Door Unobstructed state (s27). From this description, a new state machine diagram, located in Figure 16, is developed with transitions connecting the two states. This state machine provides a state, s26, that is associated with Intermediate Event e6.
- Intermediate Event e4 is the top-level fault for the Door Obstruction Fault Tree. Following the 'Fault Tree to State Machine Heuristics', this top-level fault must be converted into its own state machine. Figure 17 is by applying the transformation heuristics and applying the Fault State Machine template. The conditions under which this fault is considered resolved, when the front door is in state s4, is integrated into the state machine providing the basis for fault resolution actions.

The creation of Figure 17 provides a state, s30, associated with intermediate event e4. Now that each intermediate event in the Door Obstruction Fault Tree is associated with a state in some state machine, the fault tree diagram is modified to reflect each event-state association, shown in Figure 18.

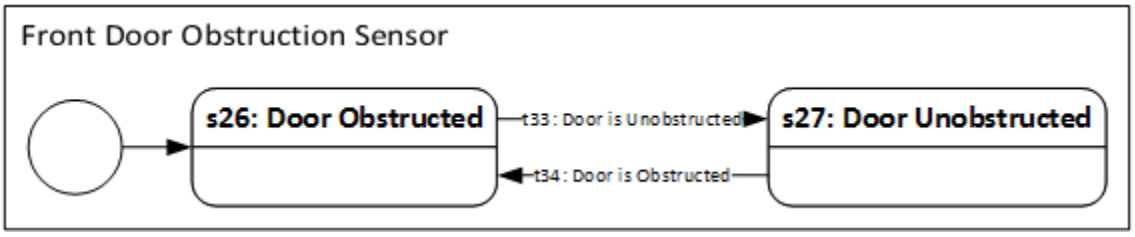

Figure 16: UML state machine diagram for the Front Door Obstruction Sensor. State s27 is active when the sensor detects an obstruction in the door while state s26 is active when the sensor does not detect an obstruction in the door. State s26 is the default state in case the sensor fails to initialize or start, resulting in increased system safety.

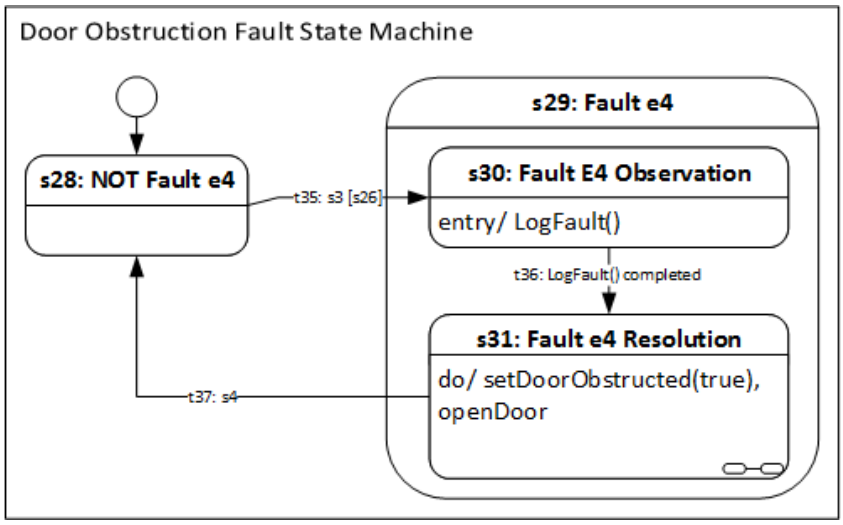

Figure 17: UML state machine diagram representing the Door Obstruction Fault Tree. Transition t35 is created using the fault tree AND gate heuristics and derived from the inputs to gate g2, e5 and e6. The logging of system details, described by s30's internal entry activity LogFault(), automatically occurs when this state is active. This allows for post-fault analysis which is helpful for system maintenance and future system development. The completion of this LogFault() function results in transition t36 and signifies that fault resolution actions may begin. Transition t37 represents the conditions under which the given fault is considered resolved; when state s4 is active.

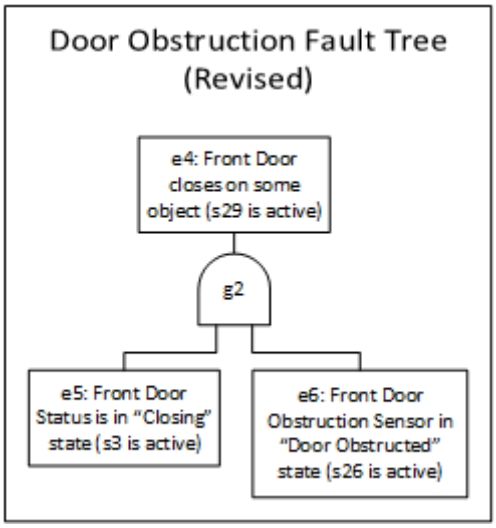

Figure 18: A revision of the Door Obstruction Fault Tree diagram where intermediate events e4, e5, and e6 are associated with some state in a state machine diagram.

Enhancing System Safety with the Safety State Machine

At this point, all state machines needed to generate the Safety State Machine are created. Figure 9, Figure 10, and Figure 11 represent the set of system's core objects, objects needed for basic system functionality. Figure 16 represents an object which affects system functionality and system safety but is not needed to achieve basic system functionality. Figure 17, derived from a fault tree for

the system, represents a hazard in the door system and provides a starting point to enhance system safety. Aggregating each of these figures into their own orthogonal region generates the Safety State Machine for the smart house front door system, shown in Figure 19.

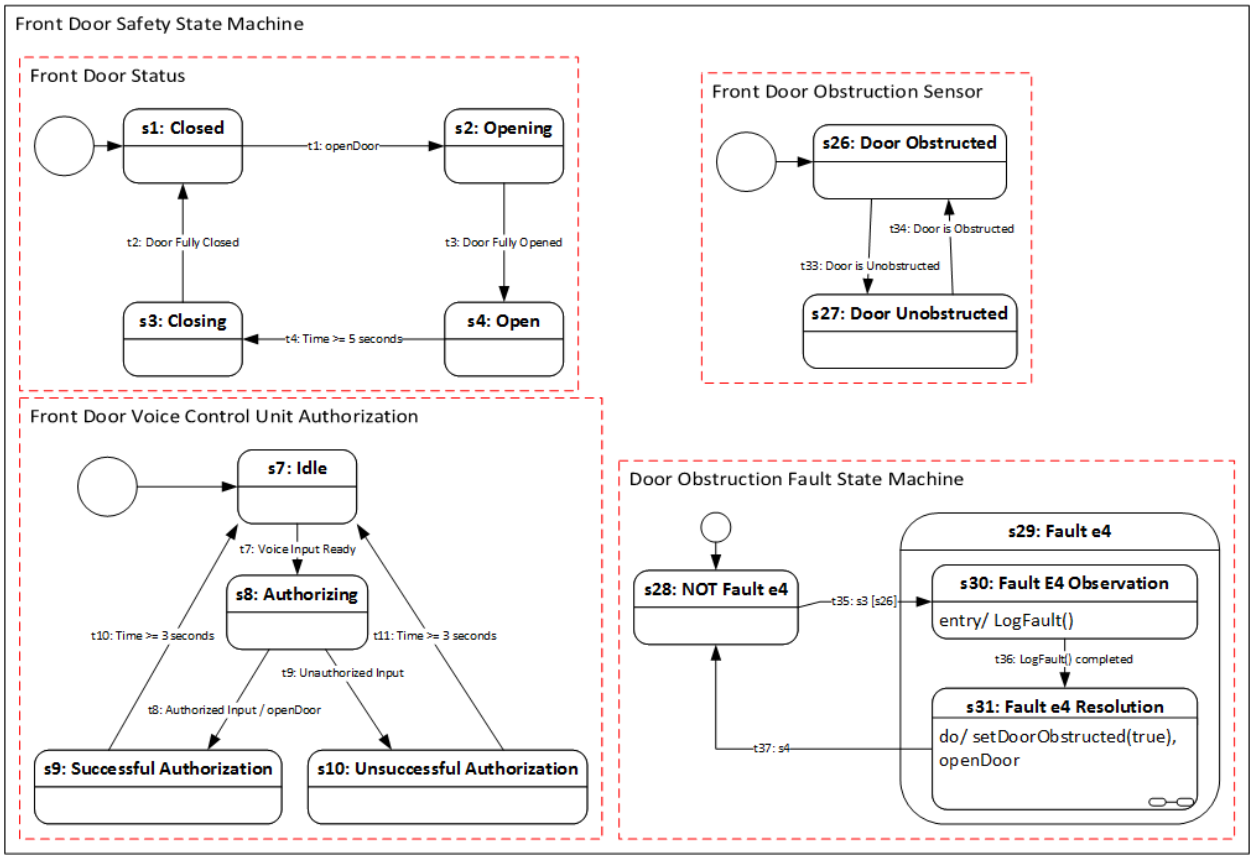

Figure 19: The completed Safety State Machine for the smart house front door system.

Enhancing System Safety

After aggregating all system influential objects into orthogonal regions, the final step of this process is to modify the state machines of these objects to prevent or otherwise mitigate the Fault X states from occurring. For this case study, state s29 is the only Fault X state. State 29's incoming transition, t35, occurs when states s3 and s26 are active. Thus the state machines for s3 and s26 are the prime candidates for modification.

There are numerous ways in which these state machines may be modified. One way to drastically reduce, if not fully eliminate, the hazard is to modify the Front Door Status state machine with three simple modifications, as follows:

1. Transition t4 gains a guard condition (s27) which lets the transition trigger only when the Door Obstruction Sensor is in a Door Unobstructed state.

2. State s3 receives an outgoing transition (t38) to state s3 which triggers when the Door Obstruction Sensor is in a Door Obstructed state.
3. State s3 receives an outgoing transition (t39) to state s3 which triggers when the door receives an openDoor signal from the Voice Control Unit.

From these modifications, shown in Figure 20 and Figure 21, the likelihood of the Door Obstruction Fault, s29, is certainly mitigated and possibly eliminated. Other fault trees can be integrated into this system's Safety State Machine to integrate additional safety specifications into the model.

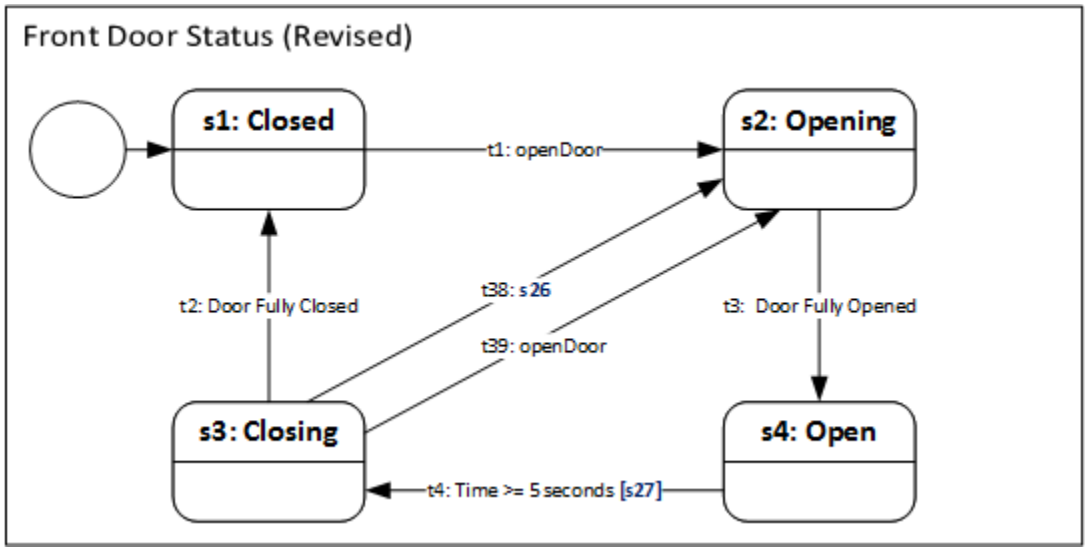

Figure 20: UML state machine diagram reflecting the revised functional specifications of the smart house front door after reducing the likelihood of s29 occurring.

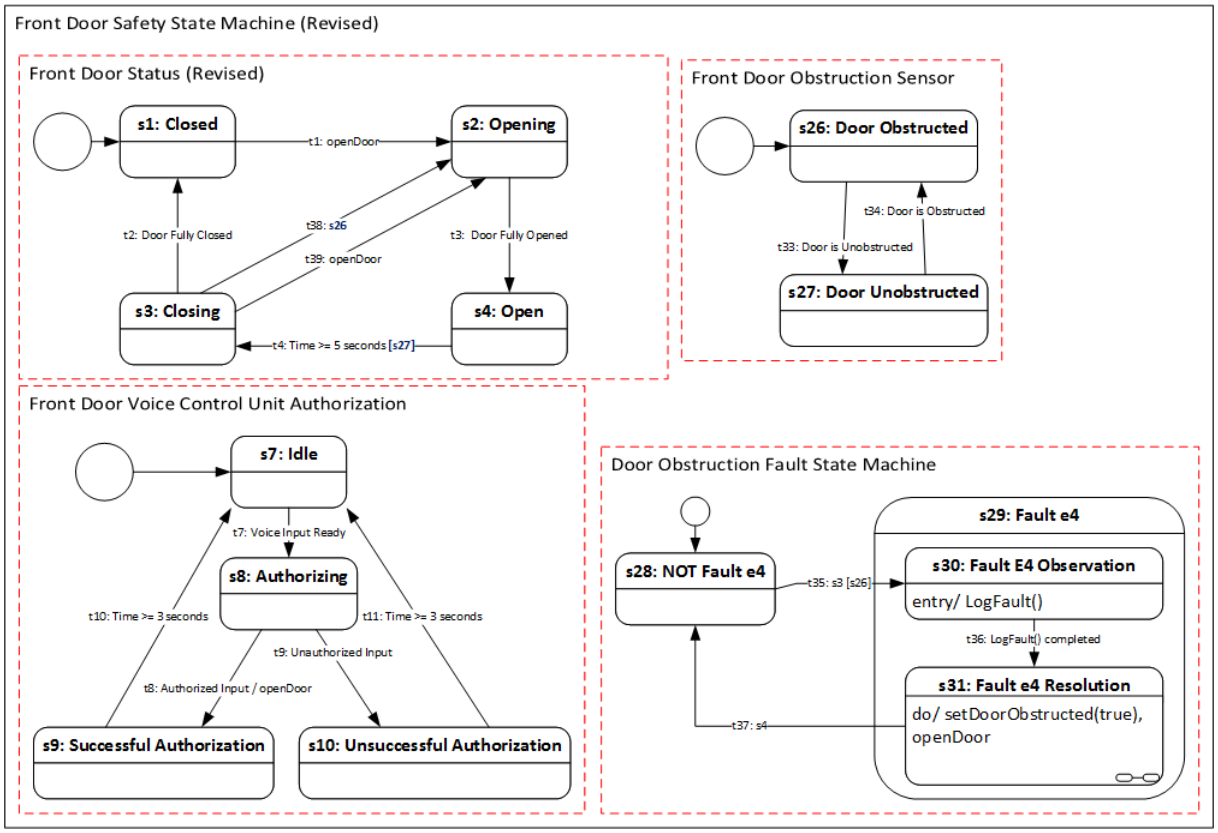

Figure 21: UML state machine diagram reflecting the revised Safety State Machine after modifying the smart house system design to mitigate the occurrence of transition t35 and consequently sate s29.

Uncovering Additional Hazards in the System

State Hierarchy Tree Truth Tables may be used to find discover additional hazards in the system. As an example, comparing the Front Door Status (Figure 9) and Front Door Primary Power Unit (Figure 10) state machines results in Table 3 which simplifies to Table 4 with Boolean algebra simplification techniques. From these tables, it is evident that the failure of the Front Door Primary Power Unit in any way poses a safety risk. As an example, should a fire within the house eliminate power, house occupants will have no way to exit the building resulting in injury or death. From this, preventative measures, such as the addition of another power source, can be developed to reduce or eliminate this safety weakness. Extending this technique to other sets of objects will help uncover other safety weaknesses existing in the system.

Row Number	Front Door Status				Front Door Primary Power Unit		Potential Hazard	Hazard Notes
	s1	s2	s3	s4	s5	s6		
1	**1**	0	0	0	**1**	0	**Yes**	Door is unmovable without power
2	**1**	0	0	0	0	**1**	No	
3	0	**1**	0	0	**1**	0	**Yes**	Door is unmovable without power
4	0	**1**	0	0	0	**1**	No	
5	0	0	**1**	0	**1**	0	**Yes**	Door is unmovable without power
6	0	0	**1**	0	0	**1**	No	
7	0	0	0	**1**	**1**	0	**Yes**	Door is unmovable without power
8	0	0	0	**1**	0	**1**	No	

Table 3: State Hierarchy Tree Truth Table comparing the Front Door Status and Front Door Primary Power Unit state machines. Due to the premise of the smart house system, rows 1, 3, 5, and 7 are hazards to those interacting with the system.

Row Number	Front Door Status				Front Door Primary Power Unit		Potential Hazard	Hazard Notes
	s1	s2	s3	s4	s5	s6		
1	**X**	**x**	**x**	**X**	**1**	0	**Yes**	Door is unmovable without power
2	**1**	0	0	0	0	**1**	No	
3	0	**1**	0	0	0	**1**	No	
4	0	0	**1**	0	0	**1**	No	
5	0	0	0	**1**	0	**1**	No	

Table 4: Simplified version of Table 3 which highlights the Front Door Primary Power Unit as a single point of failure and a safety critical object.

Section 4 – Conclusion and Future Research

In conclusion, the integration of functional specifications and safety specifications under one model helps provide a complete picture of system safety and reduce safety weaknesses resulting from using two independent models.

The State Hierarchy Tree is a diagram which describes the state hierarchy of an object's state machine. Multiple sets of State Hierarchy Tree can reveal undiscovered patterns and hazard in the system when employed in a truth table fashion, called the State Hierarchy Tree Truth Table.

The Fault Tree Conversion Heuristics provides steps to convert fault trees into a compatible state machine notation. From this, fault trees can be converted into the corresponding state machine notation and integrated into state machine diagrams.

The Safety State Machine is a model which integrates functional specifications and safety specifications. By integrating and considering all aspects which affect system functionality, the model can track the status of faults to a specified level of detail.

The utilization of the described models and heuristics provides a starting point for system safety enhancement at worst and, at best, provides a process that guides system teams to improve system safety in addition to providing a framework for testing said system safety. These models allow a system to monitor itself or allows external systems to monitor a given system. Additionally, these models are usable in fault tolerant systems where compromised components must be identified and isolated. With respect to software system safety, this models provide an excelling framework to catch error of omissions and error of commissions which often lead to faults in safety critical systems. Most importantly, this process reduces ambiguity in system architecture and design preventing the introduction of system design flaws. However, research on these models is not complete.

The aforementioned models do possess some key weaknesses and disadvantages. A prominent disadvantage is that unmeasurable object states and faults cannot be properly modeled in the Safety State Machine. As an example, fault trees with human error factors are not directly compatible with the Safety State Machine. Additionally, the fault tree conversion heuristics provided are rather informal and subjective. Implementing these heuristics into software would require the heuristics to be formalized. Finally, the Safety State Machine only properly represents designed-based safety specifications since the model is an extension of UML state machines. Other forms of UML behavioral diagrams can allow other forms of safety specifications to be observed and monitored.

Extending the models, techniques, and principles used in this paper is an area of future research. Additional areas of research include the validation of aforementioned models and heuristics, continued background research to ensure the aforementioned models and heuristics are complete, potential automation capabilities of the aforementioned models and heuristics.

Acknowledgement

This research is made possible by Dr. Eric Wong of the University of Texas at Dallas and his assisting students, the University of Texas at Dallas Computer Science Department, and the National Science Foundation (NSF).

References

[1] Object Management Group, "OMG Unified Modeling Language Superstructure," 6 August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF. [Accessed 14 June 2016].

[2] B. S. Dhillon and C. Singh, "Engineering Reliability-New Techniques and Applications," in *Chapter 4 - Fault Trees and Common Causes*, New York, John Wiley & Sons, 1981, pp. 48 - 111.

[3] International Civil Aviation Organization (ICAO), "Fault Tree Analysis (FTA) and Event Tree Analysis (ETA)," 19 November 2014. [Online]. Available: http://www2010.icao.int/SAM/Documents/2014-ADSAFASS/Fault%20Tree%20Analysis%20and%20Event%20Tree%20Analysis.pdf. [Accessed 1 July 2016].

[4] J. Xiang, F. Machida, K. Tadano, K. Yanoo, W. Sun and Y. Maeno, "A Static Analysis of Dynamic Fault Trees with Priority-AND Gates," in *2013 Sixth Latin-American Symposium on Dependable Computing*, Kawasaki, Japan, 2013.

[5] F. Ortmeier, "Formal Failure Models," *IFAC Proceedings Volumes,,* vol. 40, no. 6, pp. 145 - 150, 2007.

[6] F. Fratrik, "Challenges in Software Safety for Army Test and Evaluation," *ITEA Journal,* vol. 30, no. 3, pp. 409 - 416, 2009.

[7] O. E. Ariss, D. Xu and E. Wong, "Integrating Safety Analysis With Functional Modeling," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans,* vol. 41, no. 4, pp. 610 - 624, 2011.

[8] A. Gario and A. v. M. Andrews, "Fail-Safe Testing of Safety-Critical Systems," in *2014 23rd Australian Software Engineering Conference*, Milsons Point, NSW, 2014.

[9] H. Kim, E. Wong, V. Debroy and D. Bae, "Bridging the Gap between Fault Trees and UML State Machine Diagrams for Safety Analysis," in *2010 Asia Pacific Software Engineering Conference*, Sydney, NSW, 2011.

[10] B. Kaiser, C. Gramlich and M. Forster, "State/event fault trees—A safety analysis model for software-controlled systems," *Reliability Engineering and System Safety,* vol. 92, no. 11, pp. 1521 - 1537, 2007.

[11] K. Chen, Y.-H. Lee, E. Wong and D. Xu, "Testing for Software Safety," OSMA Software Assurance Symposium, 2007.