Software Qual J (2018) 26:3–48
https://doi.org/10.1007/s11219-015-9283-5

# Fail-safe testing of safety-critical systems: a case study and efficiency analysis

**Ahmed Gario**[1] · **Anneliese Andrews**[1] · **Seana Hagerman**[1]

Published online: 23 July 2015

**Abstract** This paper proposes an approach for testing of safety-critical systems. It is based on a behavioral and a fault model. The two models are analyzed for compatibility, and necessary changes are identified to make them compatible. Then, transformation rules are used to transform the fault model into the same model type as the behavioral model. Integration rules define how to combine them. This approach results in an integrated model which then can be used to generate tests using a variety of testing criteria. The paper illustrates this general framework using a CEFSM for the behavioral model and a fault tree for the fault model. We apply the technique to an aerospace launch system. We also investigate the scalability of the approach and compare its efficiency with integrating a state chart and a fault tree.

**Keywords** CEFSM · Finite-state machine · Safety-critical · Testing · FTA · Behavioral model · Fault model · Integration

## 1 Introduction

Safety-critical systems are systems in which a failure could lead to significant property damage, severe injuries, or loss of life. These systems have become an essential part of everyday life. For example, automobiles, medical devices, and aircraft systems are used on a daily basis. Most of these rely heavily on software. With such systems also comes the

✉ Ahmed Gario
agario@du.edu

Anneliese Andrews
andrews@cs.du.edu

Seana Hagerman
seana.l.hagerman@lmco.com

1 Department of Computer Science, University of Denver, Denver, CO, USA

exposure to risks because they may fail or may not work properly resulting in damage, injury, or death. Potential system failure is referred to as a mishap risk. For example, there is a danger that a railroad crossing system will fail, resulting in the mishap of a train colliding with a pedestrian or vehicles crossing the railway.

Testing is an important part of ensuring safe, dependable systems. Model-based testing (MBT) (Dalal et al. 1999) is common for functional testing. Models are representations of systems. They provide a functional view of the system that can be used to produce test cases without using the actual system implementation details because it is very difficult to cover all code structures especially for complex dependable systems (Utting and Legeard 2007). MBT focuses on testing the system behavior (also known as desired behavior), i.e., whether the system behaves as it should or not. However, when testing safety in safety-critical software (also known as undesired behavior), these models may not be sufficient, and more information about safety-related aspects of the system is needed. We cannot use a behavioral model to test an undesired behavior that it does not describe. Thus, using behavioral models alone to test the undesired behavior of a system will not be adequate.

Behavioral models such as unified modeling language (UML), finite-state machines (FSMs), extended finite-state machines (EFSMs), and communicating extended finite-state machines (CEFSMs) do not systematically model fault behavior. On the other hand, techniques to model failure behavior (Ericson 2005) and failure mitigation (Amberkar et al. 2001; Leaphart et al. 2005) do not address testing since these techniques are basically a static or a procedural description of the undesired behavior.

Besides the techniques used in testing software generally, testing safety-critical software systems requires analyzing the hazards beforehand by using analysis techniques such as fault tree analysis (FTA), failure modes and effects analysis (FMEA), hazard and operability analysis (HAZOP), and hazard and risk analysis (HRA) (Ericson 2005). However, there is still a gap between testing and analysis activities which negatively impacts the effectiveness of testing.

Fault tree analysis (FTA) is a safety analysis technique that is commonly used to analyze the safety of systems. It was originally designed for safety analysis in a different scientific paradigm and later used in analyzing safety-critical software (Leveson and Harvey 1983). A fault tree (FT) describes how the combination of behaviors of system components results in a hazard or a failure of a system. Although it is one of the most common techniques, it may not be suitable for software safety analysis because it is a static model that describes the overall cause of a hazard and cannot answer the questions why, when, and how the hazard occurs during software execution. On the other hand, models that are used to describe system behaviors concentrate on how a single software component behaves internally and when it interacts with other components in the system or with its environment (Ariss et al. 2011). Communicating extended finite-state machines (CEFSMs) are a type of finite-state machine used to model and test the behavior of interacting software components. Conceptually, we can think of a safety-critical system as a collection of behavioral processes that may trigger failure processes leading to specific failure events to which the safety-critical system reacts with required fail-safe mitigation actions. A common modeling technique for communicating processes is CEFSMs (Brand and Zafiropulo 1983).

Very few papers integrate the behavioral and failure models and are used for safety analysis only (Ortmeier et al. 2007; Kaiser et al. 2003; Ariss et al. 2011; Kim et al. 2010). Fewer still address testing with an integrated model (Sánchez and Felder 2003; Nazier and Bauer 2012). Unfortunately, they have only been applied to relatively small examples, so their scalability is uncertain. Compatibility between fault models and behavioral models is also an issue.

This paper extends and evaluates a technique to integrate failure and behavioral models (Gario 2014; Gario et al. 2014) for the purpose of testing both functional and fail-safe behaviors that allow for a variety of testing criteria, is scalable, and includes a step that formalizes compatibility between behavioral and failure models. The integrated model can be thought of as communicating processes where behavior and failure processes interact. We also present a case study, a launch vehicle (LV). In addition, we present an efficiency analysis of the LV and several examples from Gario et al. (2014), Gario (2014) comparing the approach to Sánchez and Felder (2003), Nazier and Bauer (2012). We also present the results of a simulation experiment to show scalability again comparing it to Sánchez and Felder (2003), Nazier and Bauer (2012).

The remainder of this paper is organized as follows. Section 2 provides a background related to behavioral and fault models and reviews the existing work. Section 3 describes the overall approach; defines CEFSM and fault tree (FT); and describes the transformation rules, the transformation procedure of FT into CEFSM, and test case generation from CEFSMs. Section 4 applies the technique to LV. Scalability and efficiency are discussed in Sect. 5. Finally, Sect. 6 provides conclusions.

## 2 Background and related work

### 2.1 Communicating extended finite-state machines (CEFSMs)

CEFSMs (Brand and Zafiropulo 1983) are an extended type of the traditional FSMs that provide data flow modeling and communication mechanisms. CEFSMs are capable of modeling and testing communicating systems as their specification includes variables, operations based on variables, and interactions between them that FSMs cannot model in a concise way (Lee and Yannakakis 1996). They are the basis for many design languages such as petri nets, message sequence chart (MSC), and specification and description language (SDL) (Li and Wong 2002). Communicating processes can often be modeled and tested as a collection of CEFSMs (Brand and Zafiropulo 1983). A variety of automated tools exist for testing CEFSM such as SDT/ITEX (Ek et al. 1997), EFTG (Bourhfir et al. 1999), and construction and analysis of distributed processes (CADP) [62], known as CAESAR/ALDEBARAN.

Some of the work in testing concurrent systems deals only with communicating finite-state machines (CFSMs) where the data part of the protocol is not considered (Bourhfir et al. 1998), while others consider data and control. Henniger et al. (2004) present an algorithm to generate a test purpose description of the behavior of a system of asynchronous CEFSMs. A test purpose can be expressed by a MSC that describes the behavior to be checked. Bourhfir et al. (1998, 2001) generate test cases for systems modeled by CEFSM. The test cases are generated for the global system by performing a complete reachability analysis and generating test cases.

Hessel and Pettersson (2007) present an algorithm for generating test suites by reachability analysis. The algorithm uses, in each step, global information about the state space to guide the analysis and to speed up termination. Kovács et al. (2002) designed methods and mutation operators to enable the automation of test selection in a CEFSM. The mutation operators create erroneous specifications that provide the basis for test case selection.

Boroday et al. (2002) use a CEFSM to generate test cases by combining the specification and fault coverage. They compute a test suite that offers specification coverage. They derive a confirming sequence from the fault model to check both states and data of each test with respect to the fault model. Li and Wong (2002) use FSMs to model behavior and events. The extension of events with variables is used to model data, while the events' interaction channels are used to model communication. The tests are generated based on a combination of behavior, data, and communication specifications. The method addresses branching coverage for data-related decision coverage and behavioral transition coverage.

## 2.2 Fault modeling and analysis

In safety-critical systems, it is essential to prevent failures so that the system can be considered safe. To make these systems low risk and fail-safe, software for safety critical systems (SCSs) must deal with the hazards identified by safety analysis. There are over 100 different hazard analysis techniques in existence. The most common analysis methods for SCSs are preliminary hazard list analysis (PHL), preliminary hazard analysis (PHA), subsystem hazard analysis (SSHA), system hazard analysis (SHA), FTA, event tree analysis (ETA), failure mode and effects analysis (FMEA), fault hazard analysis (FHA), functional hazard analysis (FuHA), sneak circuit analysis (SCA), petri net analysis (PNA), Markov analysis, hazard and operability analysis (HAZOP), cause sequence analysis, and common cause failure analysis (Ericson 2005). These techniques aid in the detection of safety flaws, design errors, and weaknesses of technical systems. The area of fault-based testing focuses mainly on faults in software (Sánchez and Felder 2003).

FTA has been used in safety-critical software. FTA was borrowed from other scientific paradigms and applied for the first time in software safety analysis in Leveson and Harvey (1983). It is a top–down deductive analysis technique used to detect the specific causes of possible hazards (Tribble and Miller 2004; Leaphart et al. 2005). The top event in a fault tree is the system hazard. FTA works downward from the top event to determine potential causes of a hazard. It uses boolean logic to represent these combinations of individual faults that can lead to the top event (Leaphart et al. 2005). FTA is a qualitative model which discloses the possible combinations of identified basic events sufficient to cause the hazard. However, it is also used in probabilistic analysis, such as frequency calculation of the hazard (Xiang et al. 2004). Each of the specific failures is reviewed, and appropriate hardware and software mitigation techniques are identified to reduce the possibility that the top event will occur (Leaphart et al. 2005).

FMEA is a bottom–up method of analyzing and evaluating safety problems in a system. The FMEA technique consists of identifying and listing all possible failure modes, evaluating the effects on the whole system for each failure mode and identifying all potential causes that may lead to each failure mode (Wang and Pan 2010). The main difference between FMEA and FTA is that the FMEA looks at all failures and their effects, whereas the FTA is applied only to those effects that are safety related and that are of the highest criticality (Czerny et al. 2005). Medikonda et al. (2011) apply FMEA and FTA to the software functions of a prototype SCS—railroad crossing control system (RCCS)—to identify possible hazardous software faults.

Due to the growth of complexity of the modern software systems, the manual verification activities are no longer practical especially for testing. MBT became a very common method for testing such complex systems. Recent techniques for model-based testing do not sufficiently take into consideration the information derived from the safety analysis such as failure mode and effect analysis (FMEA) and FTA (Kloos et al. 2011). Hence, people realized that there is a considerable gap between the safety analysis models and the behavioral models that needed to be covered. Therefore, some different approaches to integrate the fault analysis and system models were proposed and used in safety analysis and testing.

## 2.3 Integration of safety analysis and behavioral models

### *2.3.1 Safety analysis*

Safety analysis improves the probability of uncovering possible faults in safety-critical software. Ariss et al. (2011) present an approach for integrating fault-tree-based safety analysis into a functional model. They transform a FT to a statechart preserving the semantics of the fault tree and the statechart. The model shows how the system behaves when a failure condition occurs. The integrated model's purpose is for the validation of safety requirements rather than testing.

Kim et al. (2010) develop an algorithm to transform hazards of an FT into a UML statechart diagram in order to perform safety analysis. The primary events and gates of a FT are represented in a UML statechart notation. Transformation rules create a statechart diagram that can be used for safety analysis. The transformed fault tree falls into the lowest level of the composed state machine making it difficult to analyze the diagram for safety due to the indirect paths to causes of hazards.

Kaiser et al. (2003), Kaiser (2003) propose a compositional extension of the FTA technique. Each technical component in the system is represented by an extended fault tree that has, besides its basic events and gates, input and output ports. These components can be developed independently and can be integrated into a higher-level model by connecting these ports. Both qualitative and quantitative analyses can be applied on this FTA.

Kaiser (2005), Kaiser et al. (2007) proposed a combination of fault trees with an explicit state/event semantics, using a graphical notation called state/event fault trees (SEFTs). This model uses the fault tree to represent the faults which are connected to the state or event in the state/event model that describes the system behavior. However, this model is used for safety analysis. Furthermore, identifying the events for an FT and connecting them to state or event are done manually which makes the process of constructing SEFT very difficult, time-consuming, and error-prone especially for large and complicated systems. This model is used for safety analysis only. Ortmeier et al. (2007) present an approach to formally model failure modes. The functional model and the failure modes are represented as a statechart and integrated as orthogonal regions of a statechart. The integrated model is used for safety analysis only.

Many works have addressed the multi-formalism modeling of critical systems and infrastructures. These works formalized fault trees (FT), Bayesian networks (BN), and generalized stochastic petri nets (GSPN). Di Giorgio and Liberati (2011) present a dynamic bayesian network (DBN) framework for the interdependency analysis of the critical infrastructure. Three kinds of analyses on critical infrastructures based on the DBN, reliability study, an adverse events' propagation study, and a failure identification analysis can be performed with this framework. Boudali and Dugan (2005) propose a reliability modeling and analysis framework based on Bayesian network formalism to investigate timed petri nets and to find a reliable framework for dynamic systems. Montani et al. (2008) present a software tool to analyze a dynamic fault tree relying on its conversion into a dynamic Bayesian network. These works use BN and PNs to analyze the reliability and dependability of the critical systems and do not address safety testing. Buchacker et al. (1999) combined extended fault trees with stochastic petri nets for the evaluation and analysis of the model.

Bobbio et al. (2001) explore the capabilities of BN formalism in the analysis of dependable systems. They show that any FT can be directly transformed into BN to be used to improve systems dependability analysis. They show that basic inference techniques in the BN can be used to obtain classical parameters computed from the FT (i.e., reliability of the top event or of any subsystem and criticality of components). They also showed that by using BN, several restrictive assumptions in the FT methodology can be removed, dependencies among components can be taken into account, and general diagnostic analysis can be performed. Raiteri et al. (2004) extend fault trees with repairing capabilities. This extended formalism can be used to model and evaluate repair strategies for systems. It has been integrated in a multi-formalism multi-solution framework and supported by an enhanced solving techniques based on generalized stochastic petri nets (GSPN). Flammini et al. (2005) show how to use RFT advantages by evaluating the effects of different repair policies on the availability of the radio block center (RBC) of the European railway standard systems ERTMS/ETCS. Flammini et al. (2014) employed multi-formalism model to perform an availability analysis on the European railway traffic management system/European train control system (ERTMS/ETCS) system. They performed bottom–up failure model analysis from subsystem (expressed by means of generalized stochastic petri nets, fault trees, and repairable fault trees) up to the overall system model to evaluate the effect of basic design parameters on the probability of system failure modes and its availability based on its specification. Marrone et al. (2014) describe a model-driven engineering approach that proposes using UML profiles containing system analysis and test case specification capabilities, together with tool chains for model transformations and analysis in order to allow end users to focus on high-level holistic models and specification of non-functional requirements. This will guide design choices, minimize the chances of failures/non-compliance, and considerably reduce the overall assessment effort. This approach is applied to the ERTMS/ETCS. Other works such as Wada et al. (2005), Petricic et al. (2008), France and Rumpe (2007), and Petricic et al. (2009) discuss different model transformation approaches for domain-specific languages (DSL) and UML.

The CENELEC EN50128 (European Committee for Electrotechnical Standardization) guidelines for software development of safety critical system require providing a set of tests that cover 100 % of the code. However, this requirement increases the costs associated with testing phase significantly. Hence, Ghazel (2014) presents a model for formal V&V. The aim of his work was to use formal techniques to check system requirement specifications (SRS) in order to prevent specification errors. Angeletti et al. (2009) present a methodology to automatically generate test achieving the desired code coverage. The automation of the test generation, applied to some modules of the ERTMS, increased the productivity and reduced the costs of the entire software development process.

### 2.3.2 *Safety testing*

Testing safety-critical software differs from testing non-safety-critical software in many ways. Before testing safety-critical software systems, we need to conduct a safety analysis for the system to find possible safety breaches and what may cause them. We need to test desired behavior in the presence of failures. Sánchez and Felder (2003) proposed a fault-based approach for generating test cases to overcome the limitations of specification-based approaches that derive from the incompleteness of the specification of undesirable behavior, and from the tendency of specifications to focus on the desired behavior, rather than potential faults. Minimum cut sets of the FT are used to determine how undesirable states can occur in a system. These sets are transformed to equivalent statechart components. These components are integrated into the behavioral model of the system and transformed to EFSMs to flatten the hierarchical and concurrent structure of states and to eliminate broadcast communication. The problem is that flattening a statechart into an EFSM model makes it grow exponentially causing scalability problems.

Similarly, Nazier and Bauer (2012) transform fault tree events into elements of a statechart behavior model. They verify system correctness and criticality using a model checker.

## 3 Approach

Testing safety-critical systems requires testing proper behavior in the presence of failures. This means that the behavior process and failure processes interact. For MBT, we need a model that describes behavior and failure processes and their interactions, (of Fig. 1). Interaction of communicating processes is conventionally modeled by CEFSMs, while fault trees describe how certain behaviors of system components can combine to result in a system hazard or a failure. Other hazard analysis techniques do not have the same expressive capabilities.

In this approach, we propose an integration of the behavioral model with a fault model to take advantage of the two for testing. We need to generate behavioral tests to test the system behavior, generate failures at appropriate points, e.g., by injecting events into the test model (Vaos and McGraw 1998), and test proper mitigation. We also need to define coverage criteria for all three. This work is based on Gario et al. (2014), Gario (2014). We expand on this work by adding a case study and investigating scalability and efficiency.

### 3.1 Test generation process

The test generation process shown in Fig. 2 uses the behavioral model and a FT to generate test cases. It starts with the compatibility transformation step. The modified fault tree FT′ produced from this step is transformed into gate CEFSMs (GCEFSMs) according to the transformation rules. Then, the model integration step integrates the GCEFSM with the behavioral model (BM) according to the integration rules. The resultant model is the integrated communicating extended finite-state machine (ICEFSM). Test case generation methods can use this model to generate test cases based on test criteria (IC). The following subsections explain each step in more detail.

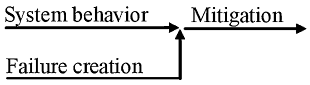
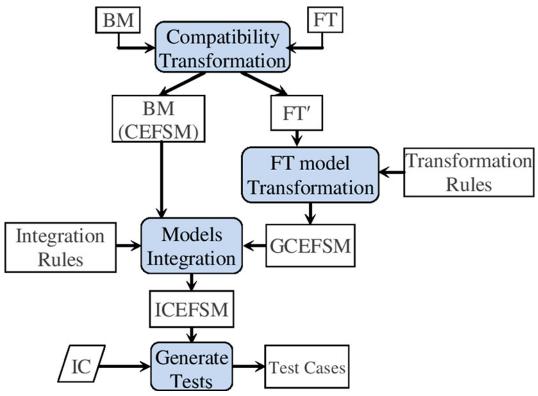


**Fig. 1** Safety-critical system behavior

**Fig. 2** Test process



## 3.2 Behavioral model: communicating EFSM (CEFSM)

A wide variety of choices for the behavioral model is possible, ranging from UML to petri nets. We chose EFSMs, because they have been widely used for testing in the embedded systems community (Sinha and Smidts 2006) including operational flight programs (Savage et al. 1997). The TestMaster tool (Software and Test 1999) uses EFSM-based test model for test generation. CEFSM has the capability of modeling a collection of communicating EFSMs. A safety-critical system is a collection of communicating behavioral processes and failure processes, and this makes CEFSM a proper modeling language to model such processes.

CEFSM has been used in modeling and testing distributed systems and network protocols. The strength of CEFSM is that it can model orthogonal states of a system in a flat manner and does not need to compose the whole system in one state as in statecharts which would make it more complicated and harder to analyze and/or test. Hessel and Pettersson (2007), Bourhfir et al. (2001), and Kovács et al. (2002) have been proposed to test systems modeled as CEFSM. These approaches are explained in more detail in Sect. 3.7.

CEFSMs can be defined as a finite set of consistent and completely specified EFSMs (Cheng and Krishnakumar 1993) that are composed via communication channels that carry input and output messages (Lee and Yannakakis 1996) along with two disjoint sets of input and output signals. A CEFSM is defined as Brand and Zafiropulo (1983):

($CEFSM = S, s_0, E, P, T, A, M, V, C$), such that:

- $S$ is a finite set of states,
- $s_0$ is the initial state,
- $E$ is a set of events,
- $P$ is a set of Boolean predicates,
- $T$ is a set of transition functions such that T: $S \times P \times E \rightarrow S \times A \times M$,
- $M$ is a set of communicating messages,
- $A$ is the set of actions,
- $V$ is a set of variables, and
- $C$ is the set of input/output communication channel used in this CEFSM.

State changes (action language): The function $T$ returns a next state, a set of output signals, and action list for each combination of a current state, an input signal, and a predicate. It is defined as:

$T(s_i,\ p_i,\ get(m_i))/(s_j,\ A,\ send(m_{j_1},\ldots,\ m_{j_k}))$ where,

- $s_i$ is the current state,
- $s_j$ is the next state,
- $p_i$ is the predicate that must be true in order to execute the transition,
- $e_i$ is the event that when combined with a predicate triggers the transition function, and
- $m_{i_1}, \ldots, m_{i_k}$ are the messages that carry the events $e_i$ for $i = 1,2,\ldots,$.

The communicating message $m_i$ is defined as:
($mId$, $e_i$, mDestination) where

- *mId* is the message identifier, and
- *mDestination* is the CEFSM the message is sent to.

An event $e_i$ is defined as: *(eId, eOccurrence, eStatus)* where,

- *eId* is the event identifier that uniquely identifies it, and
- *eOccurrence* is set to false as long as the event has never occurred. When the event occurs for the first time, *eOccurrence* is set to true and stays true, and
- *eStatus* is set to true when the event occurs and to false when it no longer applies.

Note that *eStatus* allows reoccurring events to happen multiple times (loops in the model).

CEFSMs communicate by exchanging messages through communication channels *C* that connect the outputs of one CEFSM to the input of other CEFSMs. Let *C* denote the set $\{\langle name,\ SYNC \mid ASYNC\ \rangle \mid$ for all the channels in the system$\}$ where *name* is the name of the communication channel and *SYNC* and *ASYNC* indicate that the channel is synchronous or asynchronous. A communication channel can be used by different transitions. A channel $c \in C$ can be represented as $\langle name, t, get()/send()\rangle$ where *name* is the name of the channel, $t \in T$ refers to the transition linked to this use of the channel, and *get()/send()* indicates whether this channel is an input or an output channel.

The action $a_i$ may include an assignment and mathematical operation on the variables. The predicate is a condition that must be met prior to the execution of the function. For example, $T(S_0,\ [\text{total} = 4],\ e_0)/(S_1,\ \{m_0,\ m_1\},\ (\text{total} = 0;\ \text{increment(i)}))$ describes that if a CEFSM is in a state $S_0$, receives an event $e_0$ and the predicate $total = 4$ is true at that time, it will move to the next state $S_1$ and output $m_0$ and $m_1$ after setting the *total* to zero and performing *increment(i)*. For full formal semantics, see (Brand and Zafiropulo 1983).

## 3.3 Fault tree (FT)

A FT is composed of nodes, edges, and gates. Gates are logical connectors of events, while nodes represent events, and edges connect nodes to gates. Every major failure is represented by a separate fault tree. Table 1 lists the gate types we consider here. A fault tree is analyzed either qualitatively or quantitatively (Vesely et al. 2002). Quantitative analysis is done by computing the probability of the occurrence of the root node from the probabilities of the lower-level nodes, while qualitative analysis shows the set of events that, when they happen together, contribute to cause the hazard. Qualitative analysis is applicable when integrating faults into the system model because the analysis is performed on the actual occurrence of the set of possible faults rather than on their probabilities of occurrence.

**Table 1** Fault tree gate types (Vesely et al. 2002; Ariss et al. 2011)

| Symbol | Gate | Meaning |
|---|---|---|
| $\wedge$ | AND | The gate occurs only when all its inputs occur |
| $\bar{\wedge}$ | PRIORTY AND | The gate occurs only when all its inputs occur in a specified order |
| $\vee$ | OR | The gate occurs when at least one of its inputs occurs |
| $\diamond$ | INHIBIT | The gate occurs only when the input occurs and the enabling condition is true |
| $\oplus$ | XOR | The gate occurs only when the XOR of the inputs is true |
|  | TIMING GATES | These gates occur only when event occurs and the time-out is triggered |

## 3.4 Compatibility transformation

The basic events in fault trees (leaf nodes) depend on the scope, resolution, and the ground rules (Vesely et al. 2002). The scope of the FT indicates which failure will be included and which will not, the resolution is the level of detail at which these basic events will be developed, and the ground rules include the procedure and terminology used to name these events. Often, the basic events in fault trees are informally described, i.e., in a natural language. If the resolution or the event naming does not match that of the behavioral model, which is often the case, we say these models are not compatible. Therefore, we need to make these models compatible in order to be able to integrate them. Behavioral and fault models are said to be compatible if they describe the same level of abstraction and the same events in both models have the same meaning.

The compatibility transformation procedure takes the BM and the FT as inputs and produces a FT$'$ that is compatible with the BM. The attributes of entities in FT (each leaf) and behavioral model are formalized by creating a class diagram.

1. Identify entities that have capability of failure or contributing to a failure at the behavioral model. An entity could be a state or an event.
2. For each such entity, create a *BEntityName* class with behavioral attributes.
3. Identify leaf node entities from fault trees.
4. For each such entity, create *FEntityName* class with attributes related to failure and failure condition.
5. Express *entity.failure* condition in terms of attributes of *FEntityName*.
6. Combine both *BEntityName* and *FEntityName* in *BFEntityName* by identifying attributes common to both entities such that, if values in *FEntityName* and *BEntityName* are the same, we combine the attributes, otherwise we create *Battribute* and *Fattribute*.

Figure 3 shows a *BEntityName*, a *FEntityName*, and a *BFEntityName*. The *BEntityName* contains either a state $B_S$ (a state at the behavioral model) or an event $B_E$ (an event at the behavioral model) from the behavioral model that contributes to a failure at the fault model. These events are carried in the communicating messages from the behavioral to the fault models when these models are integrated. The *FEntityName* contains a state $F_S$ (a state at the failure model) or an event $F_E$ (an event at the failure model) as described in a leaf node of a FT along with their conditions $F_C$ (if any). The *BFEntityName* contains either a combination of *BEntityName* and *FEntityName* attributes if these attributes are the

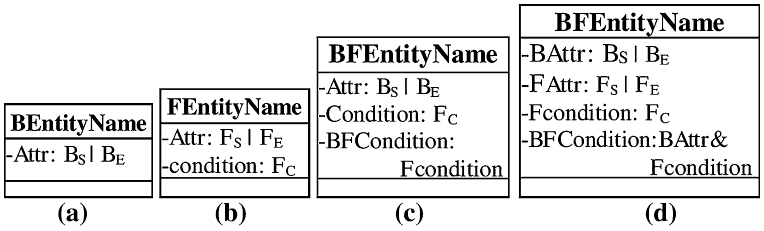
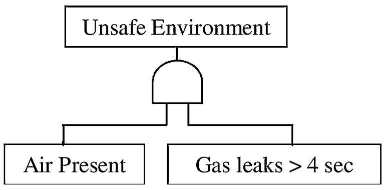


**Fig. 3** Behavioral and fault classes combination

**Fig. 4** Fault tree example



same as shown in Fig. 3c or separate *Battributes* and *Fattributes* are created as shown in Fig. 3d when the attributes of *FEntityName* and *FEntityName* are not the same.

For example, the fault tree ($\wedge$,*Air Present*, *Gas leaks* $> 4$ *s*) depicted in Fig. 4 contains two events that contribute to an unsafe environment. These events need to be made compatible with the events that have the same meaning in the behavioral model. Let us assume that the entities that have capability of failure or contributing to a failure in the behavioral model are “Air Valve” and “Gas Valve.” Therefore, *BEntityName* class named *BAirValve* will be created for the entity “Air Valve.” The attribute of this class is of type $B_S$, and its values are “Open” or “Closed.” The *FEntityName* class named *FAirValve* will be created for the leaf node “Air present.” The name of the attribute is “AirPresent,” its type is $F_S$, its values are “yes” or “no,” and the condition of this attribute *AirPresent* = *yes*. Since the names of these entities are not the same although they have the same meaning, we create a *BFAirValve* class that contains separate attributes of both *BAirValve* and *FAirValve* as described in Fig. 3d. The *BFAirValve* attributes are *BState:Open, Closed*, *FState:Airpresent:yes, no* and *BFEventCondition:AirPresent*= *yes* (Fig. 5). Also, the *BEntityName* class named *BGasValve* will be created for the entity “Gas Valve.” The attribute of this class is of type $B_S$, and its values are “Open” or “Closed.” The *FEntityName* class named *FGasValve* will also be created for the leaf node “Gas leaks > 4 s.” The name of the attribute is “Leaks,” its type is $F_S$, its values are “yes” or “no,” and the condition of this attribute *is Leaks & TimeInState >4 s*. We create a *BFGasValve* class that contains separate attributes of both *BGasValve* and *FGasValve*. The *BFGasValve* attributes are *BState:Open, Closed*, *FState:Leaks:yes, no*, *FTimeInState: 4 s*, and *BFEventCondition:Leaks & FTimeInState* $> 4$ *s* (Fig. 6).

At this point, the conditions are aggregated from the leaves of the FT to the root. The compatibility transformation is an essential step to solve the ambiguity between the events in the behavioral model and fault model. The output of this step is a $FT'$ which is described in terms of *BFClass.BFEventCondition* combined with logical operators. The compatible fault tree for this example will be: $FT'$ = ($\wedge$,*BFAirValve.BFEventCond*, *BFGasValve.BFEventCond*).

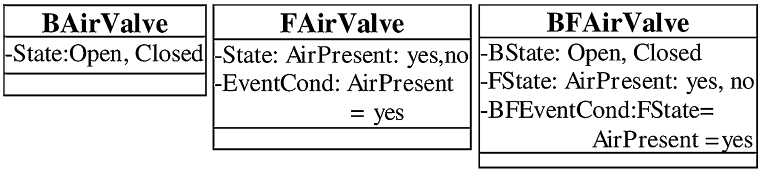

BAirValve
-State:Open, Closed
FAirValve
-State: AirPresent: yes,no
-EventCond: AirPresent = yes
BFAirValve
-BState: Open, Closed
-FState: AirPresent: yes, no
-BFEventCond:FState= AirPresent =yes

**Fig. 5** Air valve class

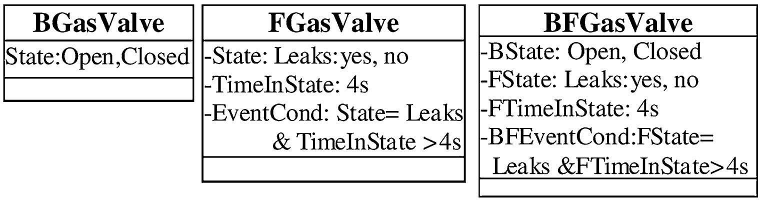


**Fig. 6** Gas valve class

## 3.5 FT′ model transformation

Events can be classified as either "transient" or "persistent" (Ortmeier et al. 2007). A transient event is an event that is reversible, i.e., it can appear and disappear completely, while the persistent event once it occurs stays. An ordinary fault tree, which statically describes a hazard, does not consider this distinction between events because this distinction would not make a difference for a static model. However, it is essential to consider the event type attribute when making a fault tree dynamic. The event type determines whether the status of the event can be "not occurred" after it had already "occurred." The change in the event status makes the integrated fault tree react according to the status of the event in the behavioral model. Note that our transformation rules allow for modeling transient events unlike the classical fault trees where all the events are persistent. However, the FT′ is a static model that describes the hazard as a specific combination of events. In order for the FT′ to be integrated into a behavioral model, it has to be dynamic, and every event at the leaf nodes has to have an equivalent event in the behavioral model. To accomplish that, we have to transform the FT′ to a CEFSM format. Every gate in the FT′ is represented as a GCEFSM. This GCEFSM represents a specific part of the failure process. Messages connect it to the behavioral process where failure process and behavioral process interact. The whole model forms a tree-like structure. The ICEFSM consists of a collection of CEFSMs that represent the behavioral model and GCEFSMs (the transformed FT) model. The communication between the behavioral model and FT′ model is achieved by sending and receiving messages between the models. The behavioral model sends messages that contains events that contribute in the failure to the fault-related GCEFSMs. These GCEFSMs, however, do not send any message back to the behavioral model because they are only used to indicate when the carried events contribute in the root node. Upon receiving those messages, the GCEFSMs at the lower level of the tree sends messages that carry "the event occurred" or "has not occurred" to the upper level GCEFSMs and so on. The output message from one GCEFSM is taken as a parameter to a generic event in the receiving GCEFSM, *e.g., event(param) = get* $(m_i)$.

**Table 2** Event sequence table

| $e_j$ | Event ($e_j$) | | TotalNoOfEvents | NoOfPositiveEvents | NoOfOccurredEvents |
|---|---|---|---|---|---|
| | $e_j$.*eOccurred* | $e_j$.*eState* | | | |
| | | | 2 | 0 | 0 |
| $e_1$ | T | T | 2 | 1 | 1 |
| $e_1$ | T | F | 2 | 0 | 1 |
| $e_2$ | T | T | 2 | 1 | 2 |
| $e_2$ | T | F | 2 | 0 | 2 |
| $e_1$ | T | T | 2 | 1 | 2 |
| $e_2$ | T | T | 2 | 2 | 2 |

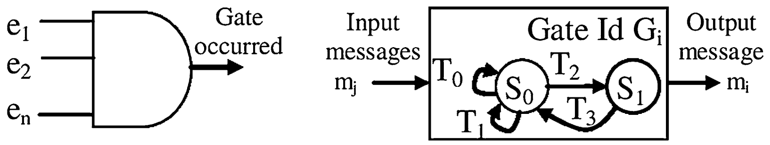


**Fig. 7** AND gate representation in FT and GCEFSM

To make the FT′ to GCEFSM transformation automatic, the representation of the FT events and gates in GCEFSM is standardized. Each gate must be given an identifier that uniquely identifies it. The output of the gate, which is an input to another gate, should carry the same identification number as the gate that outputs it. If the gate event has occurred, a message $m_i$ is sent to the receiving gate, indicating that the event has occurred. The output of each gate is an input to another gate. The GCEFSM may be in one of three conditions; it has not received any input messages so far, it received a message that says the gate event has occurred, or received a message that says the gate event has not occurred.

### *3.5.1 Transformation rules*

The transformation rules use the notation for *e,m* introduced for CEFSM in Sect. 3.2. Every gate in the FT′ is converted to an equivalent representation in CEFSM, i.e., Gate CEFSM (GCEFSM). Every GCEFSM is identified by a unique identifier $G_i$ that uniquely identifies the gate. The set of variables $V$ are:[1]

- *TotalNoOfEvents* is the total number of input events to the gate.
- *NoOfOccurredEvents* is the number of occurred events that the gate received so far.
- *NoOfPositiveEvents* is the number of occurred events whose *eStatus* is true.
- *update()* is a proposition that updates the values of *NoOfOccurredEvents* and *NoOfPositiveEvents* according to the input event $e_i$.
- *xor()* is a proposition that performs xor function on the occurred events.
- *[reset(Timer)]* is a proposition that resets the timer to zero.

[1] For readability, we omit some of the variables for the gates.

Table 2 shows the values of variables for the gates upon receiving sequence of events. Each GCEFSM consists of states and transitions that perform the same Boolean function as the gates in an FT. The difference is that in the original FT, a gate produces a single output when all the input events satisfy the gate conditions. Otherwise, no output would be produced. However, in the transformed FT, a gate has two kinds of outputs. One output is defined as the "gate occurred" and the other is defined as "gate not occurred" such that:

$$m_i = \begin{cases} \text{Gate Occurred} & \text{if} \quad G_i(e_1, e_2, ...e_k) = true, \\ \text{Gate not Occurred} & \text{if} \quad G_i(e_1, e_2, ...e_k) = false \\ & \quad and\, eOccurrence = true \\ & \quad \forall e_i, i = 1\, to\, k \end{cases}$$

For example, an AND gate = true if $G_{AND}(e_1 \cap e_2 ... \cap e_k) = true$. Each structure and behavior of each GCEFSM is predefined, and for this matter, we present the commonly used gates in this section, and the rest of the gates is presented in Sect. 7.

### *3.5.2 AND gate*

When combining some events with an AND gate, the output occurs when all the events occur. Otherwise, no output would occur. An AND gate is represented as shown in Fig. 7. It consists of two states and four transitions. State $S_0$ is the initial state, and $S_1$ is the "gate occurred" state. The transition $T_2$ will never be taken unless its predicate *NoOfOccurredEvents=TotalNoOfEvents* & *$e_i$.eOccurrence =true* & *$e_i$.eStatus = true* is true which means all the inputs are received, and their status is true. When $T_2$ is taken, the message "gate occurred" is sent to a GCEFSM that is supposed to receive it.

The transition $T_0$ is as follows:

$T_0 : (S_0, [e_j.eOccurrence = true\& \quad e_j.eStatus = true\&NoOfOccurredEvents < TotalNoOfEvents], get(m_j))/(S_0, update(events), -)$ where

1. The event get($m_j$) gets input messages from the environment or from another CEFSM. $m_i$ contains an event that could be "gate occurred" or "gate not occurred."
2. Update (events) is an action performed upon executing this transition. It updates the number of occurred events and their status based on the last input message received.
3. The predicate "[$e_j$.eOccurrence = true & $e_j$.eStatus = true & NoOfOccurredEvents < TotalNoOfEvents]" ensures that the event has occurred, and the number of inputs received so far is less than the total number of inputs, and the input status is true. Note that "gate not occurred" implies that *eOccurrence=true&eStatus=false*, while "gate occurred" implies that *eOccurrence=true&eStatus=true*.

If all the messages to this GCEFSM are received and all the events have occurred, then the transition $T_2$ will be taken.

$T_2 : (S_0, [NoOfPositiveEvents = TotalNoOfEvents\&e_j.eOccurrence = true\&e_j.e \quad Status = true], get(m_j))/(S_1, \ update(events), Send(GateOccurred))$

When this transition is taken based on the input and the predicate, it moves to state $S_1$, increments the number of inputs, and send an output message saying that the gate has occurred.

$T_1 : (S_0, [e_j.eOccurrence = true\&e_j.eStatus = false], get(m_j))/(S_0, update(events), -)$, where "-" means no output produced. When on state $S_0$ and the input message implies that the event has changed its status, the transition $T_1$ is taken. $T_1$ decrements the number of inputs and updates the status of the event from occurred to not occurred.

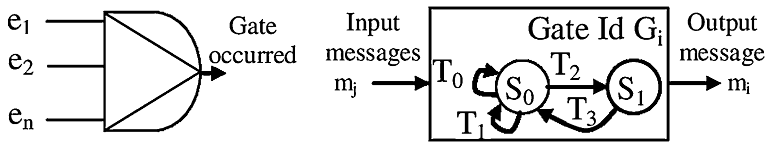


**Fig. 8** Priority And gate representation in FT and GCEFSM

$T_3 : (S_1, [e_j.eStatus = false], get(m_j))/(S_0, update(events), Send(GatenotOccurred))$

At the state S$_1$, transition T$_3$ is taken when the coming input status is false. When this transition is taken, it decrements *NoOfOccurredEvents* and *NoOfPositiveEvents*, updates the status of the input from occurred to not occurred, and sends “gate not occurred” message to the receiving gate.

### 3.5.3 Priority AND gate

As shown in Fig. 8, the priority AND gate is very similar to the AND gate in the overall structure and transitions. It differs from the AND gate in that the events have to happen in predetermined order. This difference is taken care of by manipulating the predicate condition in such a way that it considers the order of occurrence of the events. For example, if the events are ordered E$_0$, then E$_1$, they have to happen in this order so that the gate can occur. Otherwise, the gate will not occur. T$_0$ to T$_3$ are the transitions that control the priority AND gate.

$T_0 : (S_0, [NoOfPositiveEvents < TotalNoOfEvents$
$\&e_i.eStatus = true], get(m_i))/(S_0, update(events), -)$
$T_1 : (S_0, [e_j.eStatus = false\&e_j.eOccurrence = true], get(m_j))/(S_0, update\ (events), -)$
$T_2 : (S_0, [NoOfPositiveEvents = \quad TotalNoOfEvents\&ordered(e_j) = true\&e_j.eStatus = true], get(m_j))/(S_1,\ update(events), Send(GateOccurred))$
$T_3 : (S_1, [e_i.eStatus = false], get(m_i))/(S_0, update(events), Send(\ Gate\ Not\ Occurred))$

The predicate *ordered()* returns true of the input event in the message $m_i$ is received in its predefined order and returns false otherwise.

### 3.5.4 OR gate

The OR gate occurs if at least one event occurs. This gate, as shown in Fig. 9, consists of two states and four transitions. When in S$_0$ and the input message carries an event whose *eOccurrence* and *eStatus* are true (i.e., the event has occurred), T$_0$ is taken, and the OR gate occurs. In state S$_1$ and if the events in the input messages have not occurred (i.e., their *eStatus* is false), and there was only one input so far, which means this input has changed its status, then a “gate not occurred” message is sent. Otherwise, no message is sent out of this gate and only update(events) actions take place.

$T_0 : (S_0, [e_j.eStatus = true], get(m_j))/(S_1, update(inputs), Send(Gate\ Occurred))$
$T_1 : (S_1, [e_j.eStatus = false\&NoOfPositiveEvents = 0], get(m_j))/(S_1,$
$update(inputs), Send(GatenotOccurred))$
$T_2 : (S_1, [e_j.eStatus = true], get(m_j))/(S_1, update(events), -)$
$T_3 : (S_1, [e_j.eStatus = false], get(m_j))/(S_1, update(events), -)$

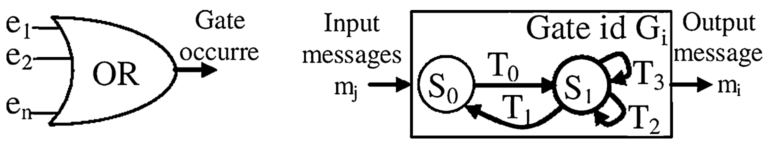


**Fig. 9** OR gate representation in FT and GCEFSM

```
Procedure FT_TO_GCEFSM (T : Tree)
{
  if (tree is null)  then return;
  for each child C of T from left to right
     do FT_TO_GCEFSM(C);
     Construct GCEFSM gate;   // Create a gate with its variables,
                              // output messages, and its ID.
   if (leaf node)
    then insert event name, event ID & Gate ID into Event-Gate table.
}
```

**Fig. 10** Transformation procedure

### 3.5.5 *Transformation procedure*

As mentioned above, the transformed GCEFSMs form a tree-like structure, and each GCEFSM gate is denoted by a unique identifier $G_i$ that uniquely identifies the gate. The transformation procedure shown in Fig. 10 takes an FT as an input and produces GCEFSMs according to a postorder tree traversal. An event–gate table is used for the integration of GCEFSMs with the behavioral model. It contains the entries for all leaf nodes of the FT and is defined as shown in Table 3. This table is constructed during the transformation of FT to GCEFSM. The leaf node event name and identifier are inserted into the table entry along with the identifier of the gate that receives this event.

## 3.6 Model integration

Before integrating the models, all messages from the behavioral model to the fault model have the form of Eq. (1). At that time, the *event id* contains the event name and attribute, and the receiving gate *id* of that event is not known yet. During the integration of both models, the event name in each message in the behavioral model is looked up in the event–gate table. If the event name and attribute in the behavioral model match those in the event–gate table, the message is modified such that it contains the *event id* $e_i$ and gate *id* $G_j$ as stated in equation (2) according to the procedure in Fig. 11.

$$m_{Bk} = (mId, EventNameAndAttribute, \_)$$

$$\textit{be a message from the BM} \tag{1}$$

$$m_{Bk} \textit{ will be modified to } (mId, e_j, G_i) \tag{2}$$

**Table 3** Event–gate table for leaf nodes

| Event name and attribute | Event ID | Gate ID |
|---|---|---|
| *event name as indicated in the FT'* | $e_i$, where $(i = 1, ..., n)$ and $e_i$ is leaf connected to $G_j$ | $G_j$ |
| Ex. temp > 10 °C | $e_1$ | $G_1$ |

```
Procedure ModelsIntegration(BM,Event-Gate Table)
{
 For every $m_{Bk}$ Do
  For every Event-Gate entries Do
   If($m_{Bk}$. EventNameAndAttribute == Event-Gate.EventNameAndAttribute)
then
          $m_{Bk}$.EventID      = Event-Gate.$e_i$
          $m_{Bk}$.mDestination = Event-Gate.$G_i$
}
```

**Fig. 11** Integration procedure

Although the compatibility transformation step may be done manually, it does not prevent the automation of the whole integration approach. In fact, it is especially important when the work has to be done manually and Sánchez and Felder (2003) would require constructing the whole integrated model manually. Note that a small number of leaf nodes in FT may produce a huge model when integrated according to Sánchez et al.'s approach (Sánchez and Felder 2003), (cf. Sect. 5.1, 19-leaf-node FT with 50-state-60-transition behavioral model would produce 70245-state-85330-transition integrated model).

In order to use CADP for larger integrated models to produce test cases, we had to automate the approach. Thus, we have implemented a front-end tool[2] that converts the fault tree into GCEFSM in the LOTOS format, converts CEFSM behavioral model, and converts it into LOTOS format and then integrating the models as we described in Sect. 3.6. The result of the front-end component which is an integrated CEFSM (ICEFSM) written in LOTOS is taken by CADP and transformed into a labeled transition system.

## 3.7 Test case generation from CEFSM model

Once the ICEFSM exists, a number of existing test generation methods for CEFSMs can be used. One approach to testing CEFSMs is to compose them all into one machine at once and using reachability analysis to generate test cases. However, this approach is impractical due to the state explosion problem and the presence of variables and conditional statements. Some work has been done in testing the behavior of concurrent systems and network protocols that were modeled using CEFSM. Hessel and Pettersson (2007) and Bourhfir et al. (1998), Bourhfir et al. (2001) use reachability analysis to generate test cases from systems modeled in CEFSMs, while Kovács et al. (2002) design methods and

[2] This front-end tool is a collaboration between the University of Denver and the University of North Dakota.

mutation operators to enable the automation of test selection in a CEFSM model. Henniger et al. (2004) generate a test purpose description of the behavior of a system of asynchronous CEFSMs. Kovács et al. (2002) use mutation to enable the automation of test selection in a CEFSM model. Boroday et al. (2002) combine specification and fault coverage to generate test cases in CEFSM models. Li and Wong (2002) propose a methodology to generate test cases from CEFSM.

## 3.8 Tool support

Construction and analysis of distributed processes (CADP) [62], formerly known as "CAESAR/ALDEBARAN Development Package," is a toolbox for communicating systems engineering. CADP's development started in 1986 by the VASY team of INRIA and the Verimag laboratory with contributions from the PAMPA team of Institute for Research in IT and Random Systems (IRISA) and the formal methods and tools (FMT) group at the University of Twente. In 2013, more than 11,000 licenses have been granted for different machines. Later, statistics are not available since academic institutions no longer have to sign a license to obtain CADP. CADP is a tool for verifying asynchronous concurrent systems. It consists of 45 tools that offer a set of functionalities that covers the design cycle of asynchronous systems such as specification, interactive simulation, rapid prototyping, verification, testing, and performance evaluation (Garavel et al. 2013). CADP can be seen as a rich set of powerful, interoperating software components. All these tools are integrated for interactive use with a graphical user interface (i.e., Eucalyptus) and for batch use with a user-friendly scripting language (SVL).

CADP [62], as supporting tool for our approach, can manage as large as $10^{10}$ explicit states, and much larger state spaces can be handled by employing compositional verification techniques on individual processes (Garavel et al. 2013). Because the textual file format that was used in the early 1990s by most verification tools is adequate for small graphs, CADP was equipped in 1994 with binary-coded graphs (BCG), a portable file format for sorting LTSs. BCG is capable of handling large state spaces (up to $10^{13}$ states and transitions CADP 2011 for 64-bit machines) (Garavel et al. 2013).

LTSs (Keller 1976) have been used to precisely represent the semantics of behavioral specifications. LTSs are used to reason about processes, such as specification, implementations, and tests. In general, an LTS provides a global monolithic description of the set of all possible behaviors of the system. It differentiates between internal and external actions. LTSs are represented by graphs of states and edges. The states represent configurations of systems, and the edges represent the moves between these configurations on the occurrence of actions. However, except for the most trivial systems, a visual representation by means of a tree or a graph is not feasible. Realistic systems would normally have billions of LTS states, therefore drawing them is not an option (Tretmans 2008).

Garavel et al. (2013) explored the distributed state spaces of a large-scale grid involving several clusters by the distributed verification tools recently added to the CADP toolbox. These experiments were intended to push the PBG machinery to its limits to study how this influences performance and scalability. They found that CADP can handle about 289,130,000 states and 542,000,000 transitions for a Dijkstra protocol of four processes. Compositional verification techniques offered by CADP are applied by Garavel et al. (2009) to overcome the state space explosion of a graph of 155,377,200 states and 371,146,000 transitions.

## 4 Application: aerospace launch system

### 4.1 Description of launch system

In this section, we demonstrate our approach with a launch system example to show the integration of multiple fault trees into CEFSMs. A launch system consists of a launch conductor, ground system, launch pad, mobile launch platform, and a launch vehicle which is comprised of a booster, upper stage, and a payload. The booster and upper stage are fueled by cryogenic fuels which can only be liquefied at extremely low temperatures. Cryogenic fuels are chosen because they generate a high specific impulse, which defines their efficiency of fuel relative to the amount consumed. A medium lift vehicle is capable of lofting a payload weighing between 4000 and 40,000 lbs. into low earth orbit. The launch controller is responsible for initiating the launch sequence and verifying the safety and security of the launch control system throughout the launch. The launch conductor communicates to the vehicle through the ground system. The ground system is physically connected to the launch vehicle via ethernet cables, serial cables, 1553 data cables, and fuel lines.

The sequence begins about 24 h before a launch when the launch conductor initiates the countdown clock. The launch conductor then clears the area of nonessential personnel using a public announcement system. The mobile launch pad is prepared for jacking. The launch conductor initiates environmental control system (ECS) on the launch pad, solicits a weather briefing, and turns on both search lights and amber warning lights. The MLP and vehicle are moved to the launch pad. Cryogenic tanking begins on the launch vehicle, and an instrumentation check is performed. A test to detect hazard gas is performed. The launch vehicle's liquid oxygen LO2 is verified as well as the upper stage's liquid hydrogen LH2. The launch conductor periodically conducts polls of the stakeholders to obtain concurrence to continue the sequence. When concurrence is received, the launch conductor initiates the chilldown procedures and flight pressures. The safe arm device (SAD) is initiated. The SAD is used to terminate the flight, should there be a problem after launch. The launch conductor commands the launch vehicle to switch to internal power and the vehicle lifts off the launch pad. Figure 4 shows the CEFSM model of the launch system including transitions, variables, events, and messages.

### 4.2 Launch system failure

The Aerospace launch system fault trees include initialization, fire, preflight, and launch fail. Initialization fail is the first fault that can occur in the system, these faults are less extreme. The initialization sequence includes connection fail, countdown clock fail, and hazard lights fail. Any of these can be mitigated with a retry before an abort command is issued. The fire fault tree sequence contains the most critical failures that could result in explosion of the system. These failures are LO2, helium, and LH2 fail. Preflight fail is the fault that can occur before a launch command is issued. Preflight fail includes battery check, initialize fuel, and battery switch fail. Launch is the final set of faults that can occur after the launch command has been issued. It includes environmental control system ECS and preflight fail. ECS includes the air conditioning failures and Nitro Purge failures. Preflight fail includes the instrument, cryotesting, and chilldown failures. The fire, pre-launch, and launch faults must be mitigated with an abort to protect the payload.

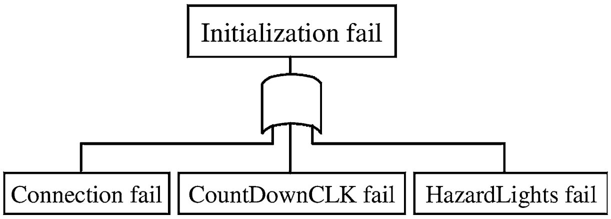


**Fig. 12** Initialization fail FT

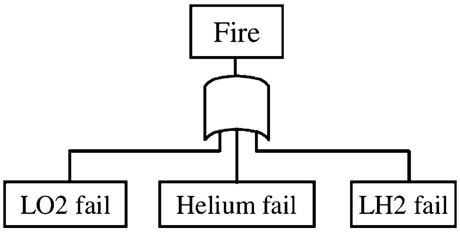


**Fig. 13** Fire occurrence FT

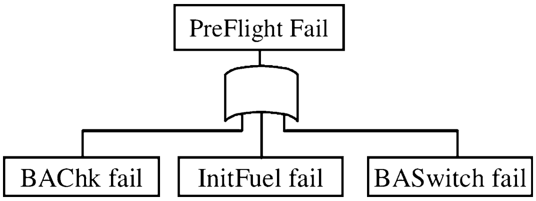


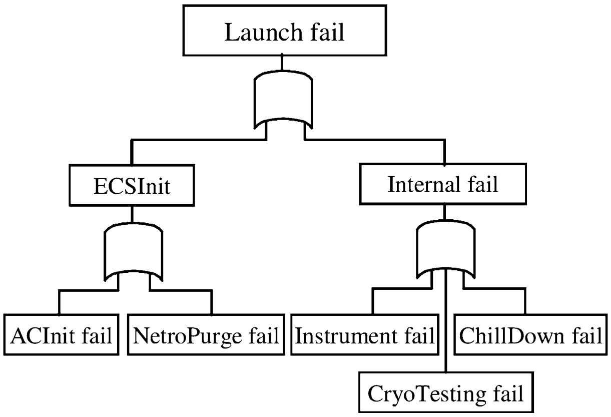
**Fig. 14** Preflight fail FT

Four launch failure occurrences are described as four FTs, one FT for each failure. The FT in Fig. 12 shows what causes the initialization failure of the launch vehicle, and the FT in Fig. 13 shows what can cause a fire and possible explosion. The preflight failure is illustrated by the FT in the Fig. 14, and the launch failure is shown in Fig. 15. These FTs will be integrated in the behavioral model shown in Fig. 4. The mitigation actions for this system are to abort. Therefore, mitigations are not applicable.

Initialization fail FT and the event description are as follows:

- Connection fail: The first step in the launch sequence requires that a connection is made between the launch vehicle, upper stage, launch platform, and ground system. This connection consists of ethernet cable to establish the ground network and 1553 cables for commanding and getting status from the launch vehicle. Failure for one of the networks to communicate would result in the launch being canceled or delayed. A retry action could be taken to attempt to establish the connection.
- CountDownClk fail: The launch vehicle and the ground system heavily rely on the countdown clock to synchronize time between them. If the countdown clock fails to start, pause or stop the result could fail to synchronize and cause a tank to be over/under filled and an explosion. If the fault were caught early on, the ground operator could retry to sync them or abort the launch.
- HazardLight fail: Hazard lights are used for safety around a launch vehicle. They consist of flashing or strobe lights to warn people in the area to keep away. The launch should not be conducted with a failure in the safety light mechanism.

**Fig. 15** Launch fail FT



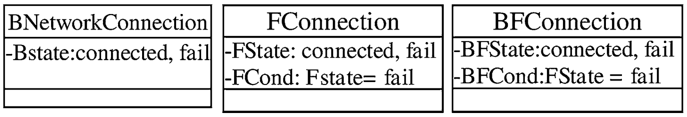


**Fig. 16** Network connection class

Fire fail FT and the event description are as follows:

- LO2 fail Liquid oxygen is cryogenic liquid oxidizer propellant for a launch vehicle. It creates a high specific impulse. The launch vehicle tank is made of thin material which is filled with L02 to pressurize it. However, LO2 will boil off and must be replenished before launch. Liquid oxygen is fed into the engine using valves. Faults associated with LO2 include: failure to pressurize, failure to top off tank, stuck valve, or defective structural integrity of the tank. The faults if not mitigated in time would result in a fire or explosion.
- Helium fail Helium is used by the upper stage to purge fuel and as an oxidizer from ground support equipment, and pre-cool liquid hydrogen. A failure from helium would result in liquid oxygen overheating and an explosion of the system.
- LH2 fail Liquid hydrogen is the upper stage cryogenic rocket propellant. It has the lowest molecular weight of any substance and burns with extreme intensity. Liquid hydrogen creates the highest specific impulse. The faults associated with liquid hydrogen include exposure to heat and leaking out of tank weld seams which would cause an explosion.

Preflight fail FT and the event description are as follows:

- BAChk fail: Battery checks are performed on the launch vehicle by the ground system. Batteries are tested for condition, state of charge is measured in volts, cell resistance is measured in ohms, and a percent of life expectancy is evaluated. Faults include: bad condition, low voltage, low cell resistance, and low life expectancy.

- InitFuel fail: Fuel initialization is the process of preparing the booster LO2 system and the upper stage LH2 system. The fuel systems are prepared by locking the valves and measuring gas pressure. Faults include low fuel pressures or bad valves.
- BASwitch fail: Prior to launching, the ground system must switch the launch vehicle from external power to internal power. This is accomplished by switching the power to the internal batteries. Internal battery failures include failure to switch, bad battery condition, low voltage, low cell resistance, and low life expectancy.

Launch fail FT and the event description are as follows:

- ACInit fail: Launch pad environmental control system air conditioning is initialized. The system fails when the air conditioning unit fails to power, or temperature is not within an acceptable range.
- NitroPurge fail: Launch pad environmental control system performs a nitrogen purge of the tanks prior to launch. Nitrogen is used to clean the impurities of the tanks. It will also displace oxygen and reduce the risk of fire or oxidation. Faults that could occur are low nitrogen pressure or stuck valve.
- Instrument fail: Prior to launch, the vehicle's instrumentation is verified by running a self or BIT (built-in test) test, and the self-test verifies the instrumentation is running properly and performs a check sum to ensure that the proper version of software is loaded. Instrumentation faults include self-test failure, checksum error, or telemetry data error.
- ChillDown fail: The chilldown procedure is used to condition fuel lines to handle the extreme cold temperatures of the cryogenic fuel. Small amounts of fuel are released from the storage tanks into the lines the feed the vehicle. Failures include: low chilldown pressure or ruptured fuel line.
- CryoTesting fail: Cryotesting is used to determine whether the vehicle will operate under extreme temperatures. This demonstration fills and drains the tanks several times. Failures include: failure to pressurize tanks and valve failure.

## 4.3 Compatibility transformation step

At this step, we create *Bclass* and *Fclass* for failure-related entities and combine the related classes according to the compatibility transformation procedure. At this step, we create *Bclass* and *Fclass*. In this example, four FTs will be integrated to the behavioral model. We start with the left most leaf node of the FT in Fig. 12. The leaf node *Connection fail* of the fault tree in Fig. 12 is related to the entity *Network Connection*. Therefore, according to the compatibility transformation rules, since the attribute of the *Bclass BNetworkConnection* and the *Fclass FConnection* is the same, they are combined in the *BFclass BFConnection*.

Next, we take its sibling, the *CountDownCLK fail*, which is represented as *FCountDownCLK* class. This *Fclass* is related to the entity *Countdown Clock* at the behavioral model which is represented as *BCountDownClock*. Therefore, they are combined into *BFCountDownCLK* class. The third leaf node in this FT is the *HazardLights fail*. This leaf node event is related to the *HazardLights On*. Therefore, we combine their related *Bclass* and *Fclass* into *BFHazardLights*. Notice that, here the values of the attributes are different, therefore, we need to include a *BAttribute* (*Bstate*) from the *BhazardLights* and *FAttribute* from the *FHazardLights* into the *BFHazardLights*.

Next, we do the compatibility transformation for the second FT Fig. 13. We start with the left most leaf node which is the event *LO2 fail* that is represented as *FLO2Chk*. This

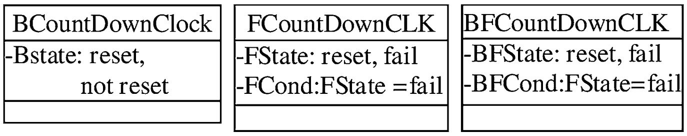


**Fig. 17** Countdown clock class

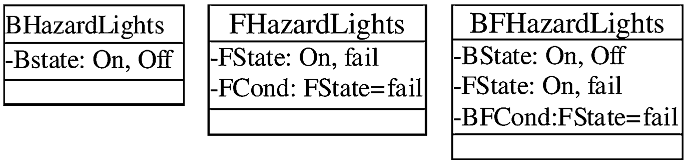


**Fig. 18** Hazard lights class

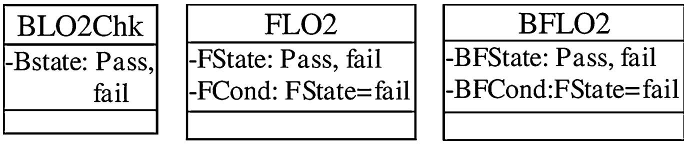


**Fig. 19** LO2 class

event is related to the entity *LO2Chk* which is represented as *BLO2Chk*. Since the attributes of these classes are the same, we combine them into *BFLO2* as shown in Fig. 19. The next event to transform is the *Helium fail*. It is related to the *HeliumChk* entity, and both have the same attributes. Therefore, they are combined into *BFHelium* (Fig. 20). The next event in this FT is the leaf node *LH2 fail*. It is related to the *LH2Chk* entity at the behavioral model. The *LH2Chk* and *LH2 fail* are represented as *BLH2Chk* and *FLH2*, respectively. Since these events have the same attributes, they are combined in *BFLH2*.

Having finished all leaf node events in the FT in Fig. 13, we start with the left most leaf node event of the *PreFlight fail* FT (Fig. 14), which is *BAChk fail* that is represented as *FBAChk*. It is related to the entity *BatteryChk* which is represented as *BBatteryChk* class. These classes are combined in *BFBatteryChk* class (*cf* Fig. 22). *InitiFuel fail*, represented as *FInitFuel*, is related to *InitiateFueling* entity which is represented as *BInitiateFueling*. As shown in Fig. 23, these two classes are combined in *BFInitFuel*. Figure 24 shows the combination of the event *BASwitch fail* and *IntBattery*. These two events are represented in *BInternalBattery* and *FBSwitch* classes, respectively.

Next, we analyze the fault tree for launch fail (Fig. 15) starts with the left most leaf node event which is *ACInit fail*. This event is represented as *FACInition* class and is related to the entity *Air Conditioning* which is also represented as *BAirCondition*. The attributes of these classes are the same so they are combined in *BFACInitiation* class as shown in Fig. 25. The next leaf node event is *NitroPurge fail* which is related to the entity *NitrogenPurge* at the behavioral model. The *NetroPurge fail* is represented as *FNitrogenPurge* class, and the *BNitrogenPurge* is represented as *NitrogenPurge* class. These two classes are combined in *BFNitrogenPurge* as shown in Fig. 26. Next, we take the event *Instrument*

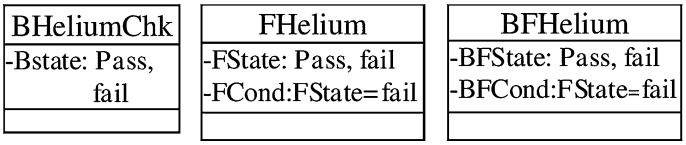


**Fig. 20** Helium class

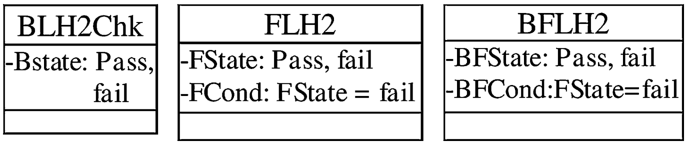


**Fig. 21** LH2 class

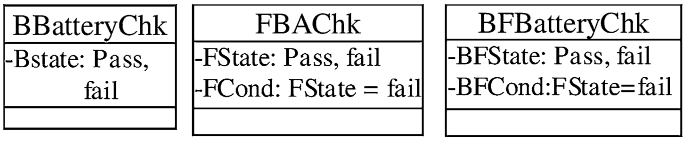


**Fig. 22** Battery class

*fail*. This event is related to the *INSTChk* entity. They are represented as *FInstrument* and *BINSTChk*, respectively, and combined into *BFInstrument* as illustrated in Fig. 27.

The event *CryoTesting fail* is transformed next. This event is represented in *FCryoTesting* and is related to the *CryoTesting* entity which is also represented as *BCryoTesting* class. The combination of these two classes is the *BFCryoTesting* class which can be seen in Fig. 28. Finally, we transform the event *ChillDown fail* to be compatible with the entity *ChillDown*. They are represented as *FChilldown* class and *BChilldown* class and are combined in *BFChilldown* class as Fig. 29 shows.

After the compatibility transformation procedure is finished, the fault tree of the initialization failure Fig. (12) is represented as:

FT′ =(∨, (*BFConnection.BFCond*,*BFCountDownCLK. BFCond*,*BFHazardLight.BFCond*)).

The FT in Fig. 13 is presented as:

FT′ =(∨,(*BFLO2.BFCond*,*BFHeluim.BFCond*,*LH2.BFCond*))

The FT in Fig. 14 is presented as:

FT′ =(∨,(*BFBatteryChk.BFCond*,*BFInitFuel.BFCond*,*BFIntBatSwitch. BFCond*))

The FT in Fig. 15 is presented as:

FT′ =(∨,(∨,*BFACInitia tion.BFCond*,*BFNitrogenPurge.BFCond*),(∨,(*BFInstrument. BFCond*,*BFCryoTesting.BFCond*,*BFChilldown.BFCond*)))

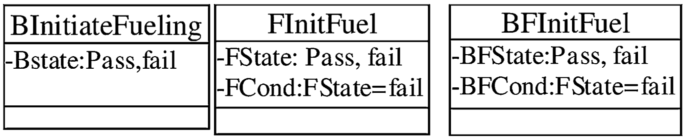


**Fig. 23** Initiating fueling class

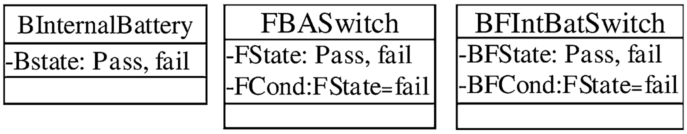


**Fig. 24** Battery switching class

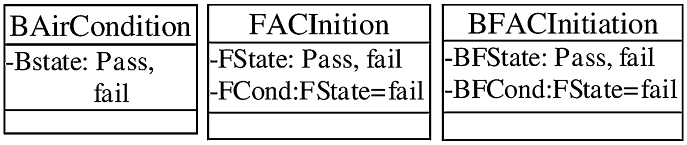


**Fig. 25** Air conditioning initiation class

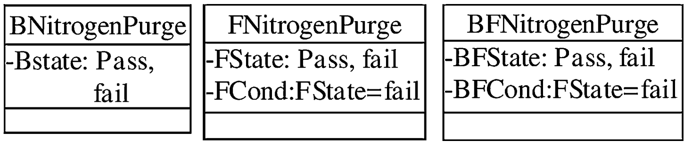


**Fig. 26** Nitrogen class

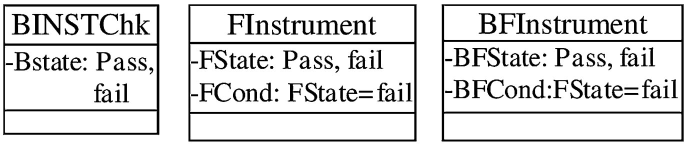


**Fig. 27** Instruments class

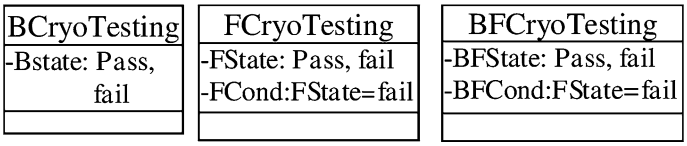


**Fig. 28** Cryo class

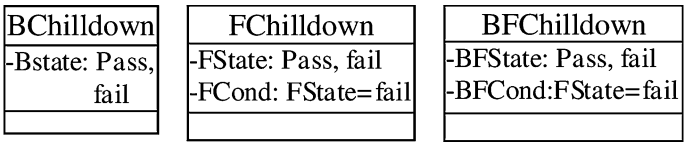


**Fig. 29** Childown class

## 4.4 Fault tree transformation

The fault CEFSM is constructed according to a tree postorder traversal. Each FT is read gate by gate starting from the root node until we reach the left most leaf node. The transformation starts with the left most leaf of the FT. The events are described in terms of class diagram states and events as shown in the compatibility transformation step. We start with the Initialization fail FT of Fig. 12. We traverse this FT from the root to the left most leaf node, the *connection fail*. Since this is a leaf node, we give it an event ID and the gate ID, and insert it in the event–gate table. Each event and the gate ID is assigned a unique sequential ID according to their appearance in the table. The next event is the *CountDownCLK fail* as expressed in the condition determined by the compatibility step, and the third is *HazardLights fail*. These events are shown in Table 5. This FT contains only one gate, and its GCEFSM can be seen in Fig. 30.

The next FT to transform into GCEFSM is the fire occurrence FT, Fig. 13. We start with the left most leaf node which is *LO2 fail*. Since it is a leaf node, we give it an event ID and the gate ID, and insert it in the event–gate table. Then, we take its siblings from left to right. The next sibling is the *Helium fail* event. We give it an event ID and insert it in the event–gate table. Then we take the last sibling and apply the same steps on it. At this point, all the leaf nodes of this FT are processed, we create the gate and give it a Gate ID. Figure 31 shows the GCEFSM for this FT, and Table 6 shows the event–gate table after the transformation of this FT.

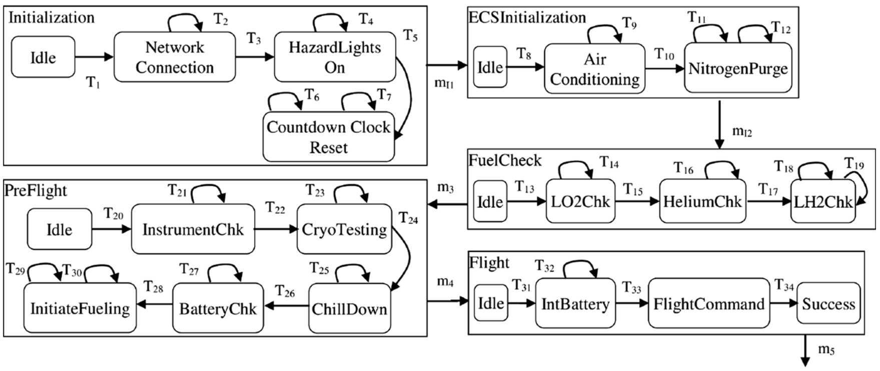
The preflight fail FT in Fig. 14 is then transformed following the same procedure. The first leaf node is *BAChk fail*. We give this event an event ID and inset it into the event–gate table with the gate ID that this event is linked to. We take its sibling, *InitFuel fail,* and we give it an event ID and inset it into the event–gate table. We do the same thing with the last event in this FT, which is the *BASwitch fail*; and then we create the gate. The GCEFSM of this FT is shown in Fig. 32, and the event–gate table is shown in Table 7.

The *Launch fail* FT is then transformed to an equivalent GCEFSM. The leaf node *ACInit fail* is read first, given an event ID and inserted into the table. Second, the event *NitroPurge fail* is read and given an event ID and inserted into the event–gate table. Then, the GCEFSM OR gate is created. This step is shown in Fig. 33.

Next, we take the leaf node *Instrument fail*, *CryoTesting fail*, and *ChillDown fail* one after another, and we take the same action for each one. At this point, all the leaf nodes of this FT are read, and all the related gates are transformed into GCEFSMs. Figure 34 shows the GCEFSM for this gate, and Table 8 shows the contents of the event–gate table at this point. Next, the gate is transformed into GCEFSM.

Then, we take the gate at the upper level of this FT. This gate is an OR gate. We transform it and assign the events from the lower-level gates. Figure 35 shows the whole *flight fail* GCEFSM.

**Table 4** CEFSM model for a launch system



$\mathbf{T}_{1}$:(Idle,[startSequence=True],startConnection)/(NetworkConnection,-)

$\mathbf{T}_{2}$:(NetworkConnection,[ConnectionConfirmed=false|time-out > =30000])/(NetworkConnection,-,send(mf$_2$ “NetworkConnectionfail”))

$\mathbf{T}_{3}$:(NetworkConnection,[ConnectionConfirmed=True],TurnLightsOn)/(HazardLightsOn,-,-)

$\mathbf{T}_{4}$:(HazardsightsOn,[AllHazardLighsOn=false],)/(HazardLightsOn,,send(mf$_4$“HazardLightsfail”))

$\mathbf{T}_{5}$:(HazardLightsOn,[AllHazardLighsOn=true],ResetClock)/(CountDownClockReset,-,-)

$\mathbf{T}_{6}$:(CountDownclockRwset,[ClkError=true],)/(CountDownClockReset,-,send(m$_{l1}$”startAC”))

$\mathbf{T}_{7}$:(CountDownclocLReset,[ClkError=false],)/(CountDownClockReset,-,send(mf$_7$“CLKFail”))

$\mathbf{T}_{8}$:(Idle,get(startAC))/(AirConditioning,-,-)

$\mathbf{T}_{9}$:(AirConditioning,[ACError=true],purge)/(AirConditioning,,send(mf$_9$”ACError”))

$\mathbf{T}_{10}$:(AirConditioning,[ACError=false],purge)/(NitrogenPurge,-,-)

$\mathbf{T}_{11}$:(NitrogenPurge,[ECSError=false])/(NitrogenPurge,-,send(mf$_{11}$”fuelcheckFail”))

$\mathbf{T}_{12}$:(NitrogenPurge,[ECSError=ture])/(NitrogenPurge,,send(mf$_{l2}$“LH2Chk”))

$\mathbf{T}_{13}$:(Idle,,get(m$_2$”fuelcheck”)/(LO2Chk,-,-)

$\mathbf{T}_{14}$:(LO2Chk,[LO2leak=true|LO2PressureOk=flase])/(LO2Chk,,send(mf$_{14}$”LO2fail”))

$\mathbf{T}_{15}$:(LO2Chk,[LO2leak=false&LO2PressureOK=true])/(HeliumChk,-,-)

$\mathbf{T}_{16}$:(HeliumChk,[Heliumleak=true|HeliumPressureOK=false])/(HeliumChk,,send(mf$_{16}$”Heliumfail”))

$\mathbf{T}_{17}$:(HeliumChk,[Heliumleak=false&HeliumPressureOK=true])/(LH2Chk,-,)

$\mathbf{T}_{18}$:(LH2Chk,[LH2leak=true|LH2PressureOk=false])/(LH2Chk,-,send(mf18”Heliumfail”))

$\mathbf{T}_{19}$:(LH2Chk,[LH2leak=false&LH2PressureOK=true])/(LH2Chk,-,send(m$_{l3}$”PreFlight”))

$\mathbf{T}_{20}$:(Idle,,get(m3”PreFlight”)/(INSTChk,-,-)

$\mathbf{T}_{21}$:(INSTChk,[ChkcksumOK=false|LaunchConductCommOk=false])/(INSTChk,-,send(mf$_{21}$”Instrufail”))

$\mathbf{T}_{22}$:(INSTChk,[ChecksumOk=true&LaunchConductCommOk=true])/(CryoTesting,-,-)

$\mathbf{T}_{23}$:(CryoTesting,[IntTempOK=false|IntPressureOk=false])/(CryoTesting,,send(mf$_{23}$”INSTfail”))

$\mathbf{T}_{24}$:(CryoTesting,[IntTempOK=true&IntPressureOk=true])/(ChillDown,-,-)

$\mathbf{T}_{25}$:(ChillDown,[IntTempOK=false|InterPssurOK=false])/(ChillDown,-,send(mf$_{25}$”ChillDownfail”))

$\mathbf{T}_{26}$:(ChillDown,[IntTemIOK=true&IntPressurOK=true])/(BatteryChk,-,-)

$\mathbf{T}_{27}$:(BatteryChk,[BatteryPresent=false|PowerLevelOK=false |BatteryLifeOK=false])/(BatteryChk,-,send(mf27”Batteryfail”))

$\mathbf{T}_{28}$:(BatteryChk,[BatteryPresent=true&PowerLevelOK=true&BatteryLifeOK=true])/(InitiatFueling,-,-)

$\mathbf{T}_{29}$:(InitiateFueling,[TankPressureOK=false|FuelLevelOK=false|TankTempOK=false])/(InitiatFueling,,send(mf$_{29}$”Fuelingfail”))

$\mathbf{T}_{30}$:(InitiateFueling,[TankPressureOK=true&FuelLevelOK=true&TankTempOK=true])/(InitiateFueling,-,send(m$_{l4}$”Flight”))

**Table 4** continued

| |
|---|
| $T_{31}$:(Idle,,get($m_4$" Flight")/(InternalBattery,-,-) |
| $T_{32}$:(InternalBattery,[SwitchToBatteryOK=false\|PowerLevelOK=false])/(InternalBattery,-,send($mf_{32}$"InternalBatteryfail")) |
| $T_{33}$:(InternalBattery,[SwitchToBattOK=true&PowerLevelOK=true])/(FlightCommand,-,-) |
| $T_{34}$:(FlightCommand,[StartFlight=true],StartFlight)/(Success,-,send($m_5$)) |

**Table 5** Event–gate table after transforming FT in Fig. 12

| Event name and attribute | Event ID | Gate ID |
|---|---|---|
| *BFConnection.BFCond* | $e_{B1}$ | $G_1$ |
| *BFCountDownCLK.BFCond* | $e_{B2}$ | $G_1$ |
| *BFHazardLight.BFCond* | $e_{B3}$ | $G_1$ |

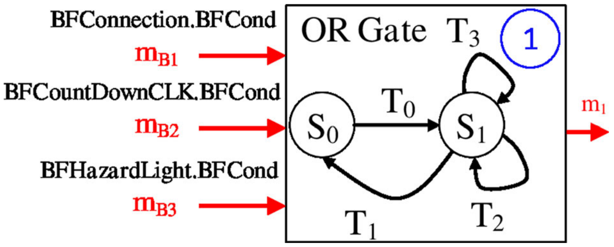


**Fig. 30** GCEFSM for the FT in Fig. 12

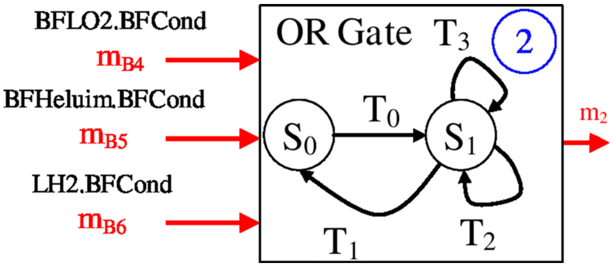


**Fig. 31** GCEFSM for fire occurrence FT in Fig. 13

## 4.5 Model integration

After all fault trees are transformed to GCEFSMs, we start integrating them into the behavioral model. At this point, every message in the BM contains an event name that is related to an event in one of the fault trees. We check the class diagram and the event–gate table to find the event ID and the gate ID for the event. These event ID and gate ID are

**Table 6** Event–gate table after transforming FT in Fig. 13

| Event name and attribute | Event ID | Gate ID |
|---|---|---|
| *BFConnection.BFCond* | $e_{B1}$ | $G_1$ |
| *BFCountDownCLK.BFCond* | $e_{B2}$ | $G_1$ |
| *BFHazardLight.BFCond* | $e_{B3}$ | $G_1$ |
| *BFLO2.BFCond* | $e_{B4}$ | $G_2$ |
| *BFHeluim.BFCond* | $e_{B5}$ | $G_2$ |
| *LH2.BFCond* | $e_{B6}$ | $G_2$ |

**Table 7** Event–gate table after transforming FT in Fig. 14

| Event name and attribute | Event ID | Gate ID |
|---|---|---|
| *BFConnection.BFCond* | $e_{B1}$ | $G_1$ |
| *BFCountDownCLK.BFCond* | $e_{B2}$ | $G_1$ |
| *BFHazardLight.BFCond* | $e_{B3}$ | $G_1$ |
| *BFLO2.BFCond* | $e_{B4}$ | $G_2$ |
| *BFHeluim.BFCond* | $e_{B5}$ | $G_2$ |
| *LH2.BFCond* | $e_{B6}$ | $G_2$ |
| *BFBatteryChk.BFCond* | $e_{B7}$ | $G_3$ |
| *BFInitFuel.BFCond* | $e_{B8}$ | $G_3$ |
| *BFIntBatSwitch.BFCond* | $e_{B9}$ | $G_3$ |

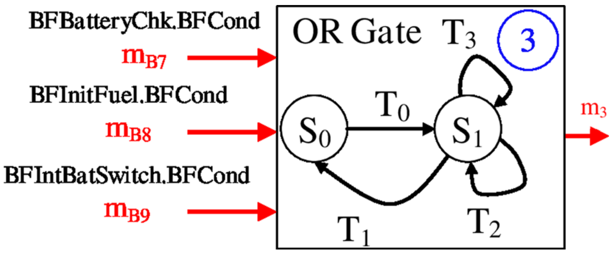


**Fig. 32** GCEFSM for the preflight failure FT in Fig. 14

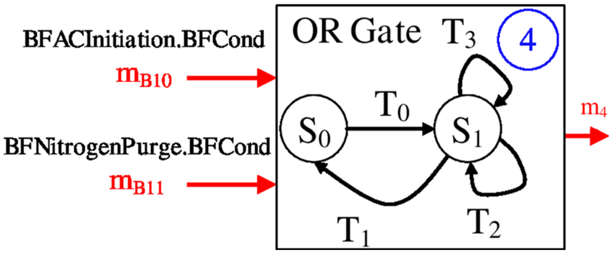


**Fig. 33** GCEFSM for an OR gate in Fig. 15

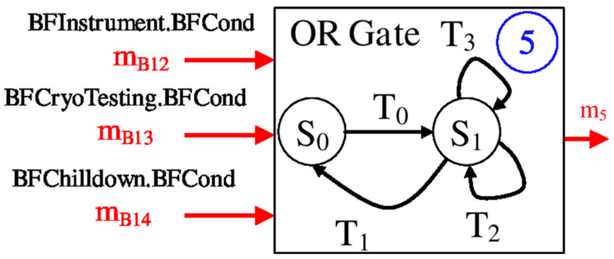


**Fig. 34** GCEFSM for the second Or gate in Fig. 15

**Table 8** Event–gate table after transforming FT in Fig 15

| Event name and attribute | Event ID | Gate ID |
|---|---|---|
| BFConnection.BFCond | $e_{B1}$ | $G_1$ |
| BFCountDownCLK.BFCond | $e_{B2}$ | $G_1$ |
| BFHazardLight.BFCond | $e_{B3}$ | $G_1$ |
| BFLO2.BFCond | $e_{B4}$ | $G_2$ |
| BFHeluim.BFCond | $e_{B5}$ | $G_2$ |
| LH2.BFCond | $e_{B6}$ | $G_2$ |
| BFBatteryChk.BFCond | $e_{B7}$ | $G_3$ |
| BFInitFuel.BFCond | $e_{B8}$ | $G_3$ |
| BFIntBatSwitch.BFCond | $e_{B9}$ | $G_3$ |
| BFACInitiation.BFCond | $e_{B10}$ | $G_4$ |
| BFNitrogenPurge.BFCond | $e_{B11}$ | $G_4$ |
| BFInstrument.BFCond | $e_{B12}$ | $G_5$ |
| BFCryoTesting.BFCond | $e_{B13}$ | $G_5$ |
| BFChilldown.BFCond | $e_{B14}$ | $G_5$ |

inserted into the message at the BM. The event "NetworkConnection fail" in the message $mf_2$ is represented in the class diagram as *BFConnection.BFCond*. This event is looked up inside the event–gate table to obtain its event ID $(e_{B1})$ and the gate ID $(G_1)$ for the gate that receives this event. The message is modified as $(m_{B1}, e_{B1}, G_1)$.

The event "HazardLights fail" in the next message $mf_2$ is represented as *BFCountDownCLK.BFCond* in the class diagram. This event is looked up in the event–gate table to obtain its event ID and the gate ID for the gate that receives this event. They are $e_{B2}$ and $G_1$, respectively. The message is modified as $(m_{B2}, e_{B2}, G_1)$. The next message to be modified is the message that carries the event "CountDownCLK fail." This event is represented as *BFCountDownCLK.BFCond*. The event ID and the gate ID for this event are $e_{B3}$ and $G_1$, respectively. The modified message will look like $(m_{B3}, e_{B3}, G_1)$. These events happen to be for the same FT, and this FT has only these three events as leaf nodes which means that the first FT is integrated.

The next event to check is "ACError" in the message $mf_9$. This event is represented in the class diagram as *BFACInitiation.BFCond*. This event is looked up in the event–gate table to obtain its event ID and the gate ID this event is an input to which are $e_{B10}$ and $G_4$. The message will be $(m_{B10}, e_{B10}, G_4)$. The next event from the BM is "fuelcheck Fail" in the message $mf_{12}$. This event is represented as *BFNitrogenPurge.BFCond*. Its event ID and

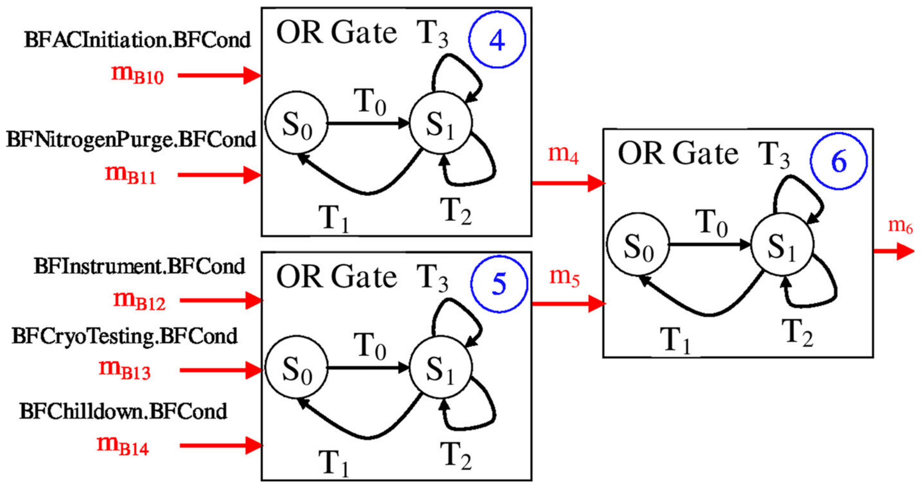


**Fig. 35** GCEFSM for flight fail FT in Fig. 15

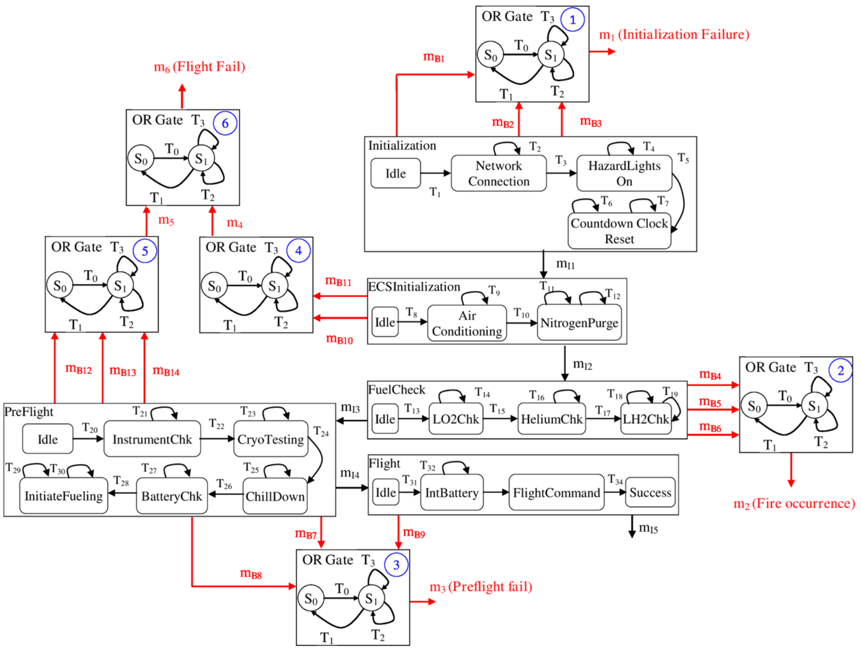
gate ID are looked up in the event–gate table. This message will be modified as $(m_{B11}, e_{B11}, G_4)$.

The event ”LO2 fail” in the message $mf_{14}$ which is represented as *BFLO2.BFCond* is looked up in the event–gate table for the event ID and the gate ID. The message will be modified to be $(m_{B4}, e_{B4}, G_2)$. The “Helium fail” event in the message $mf_{16}$ is looked up in the event–gate table to obtain its event ID and the gate ID for the gate that receives this event. This event is represented as *BFHelium.BFCond*. The message is modified to be $(m_{B5}, e_{B5}, G_2)$. The event ”LH2fail” in the message $mf_{18}$ at the behavioral model is taken next. According to the compatibility transformation, this event is represented as *LH2.BFCond*; and from the event–gate table, its event ID is $e_{B6}$ and the gate ID for the gate that receives this event is $G_2$. Therefore, the message is modified as $(m_{B6}, e_{B6}, G_2)$. The next event from the begavioral model to check is “Instrufail” in the message $mf_{21}$. This event is represented as *BFInstrument.BFCond*. Its event ID and gate ID are looked up in the event–gate table, and they are $e_{B12}, G_5$. Therefore, the message is modified as $(m_{B12}, e_{B12}, G_5)$.

The integration procedure continues for all remaining messages from the behavioral model. These messages are: the message $mf_{23}$ carrying the event “cryoTestingfail” becomes $(m_{B13}, e_{B13}, G_5)$, the message $mf_{25}$ carrying the event “ChillDownfail” becomes $(m_{B14}, e_{B14}, G_5)$, the message $mf_{27}$ carrying the event “Batteryfail” becomes $(m_{B7}, e_{B7}, G_3)$, the message $mf_{29}$ carrying the event “initiateFueling fail” becomes $(m_{B8}, e_{B8}, G_3)$, and the message $mf_{32}$ carrying the event “InternalBatteryfail” becomes $(m_{B9}, e_{B9}, G_3)$. Once all the messages are assigned to their GCEFSM destinations, the behavioral model and the fault trees are integrated. Figure 9 shows the integrated model ICEFSM.

The transformed system shown in Fig. 9 forms a graph to which suitable coverage criteria can be applied. The FT gates that are directly connected to the behavioral model receive messages from the behavioral model and act accordingly. The messages $m_1$ to $m_5$ represent the global transitions between the GCEFSMs for the FTs, while $m_{I1}$ to $m_{I5}$ represent the messages between the components of the behavioral model and $m_{B1}$ to $m_{B14}$ represent the communicating messages between the behavioral and fault models. If we

**Table 9** ICEFSM model for a launch system



**T**$_1$:(Idle,[startSequence=True],startConnection)/(NetworkConnection,-)

**T**$_2$:(NetworkConnection,[ConnectionConfirmed=false|time-out > =30000])/(NetworkConnection,-,send(m$_{B1}$))

**T**$_3$:(NetworkConnection,[ConnectionConfirmed=True],TurnLightsOn)/(HozardLightsOn,-,-)

**T**$_4$:(HazardsightsOn,[AllHazardLighsOn=false],)/(HazardLightsOn,,send(m$_{B2}$))

**T**$_5$:(HazardLightsOn,[AllHazardLighsOn=true],ResetClock)/(CountDownClockReset,-,-)

**T**$_6$:(CountDownclockReset, [ClkError = true],)/ (CountDownClockReset,send(m$_{I1}$))

**T**$_7$:(CountDownclocLReset, [ClkError = false],)/ (CountDownClockReset,-,send(m$_{B3}$))

**T**$_8$:(Idle,get(m$_{I1}$))/(AirConditioning,-,-)

**T**$_9$:(AirConditioning,[ACError=true],purge)/(AirConditioning,-,send(m$_{B10}$))

**T**$_{10}$:(AirConditioning,[ACError=false],purge)/(NitrogenPurge,-,-)

**T**$_{11}$:(NitrogenPurge,[ECSError=false])/(NitrogenPurge,-,send(m$_{B11}$))

**T**$_{12}$:(NitrogenPurge,[ECSError=ture])/(NitrogenPurge,-,send(m$_{I2}$))

**T**$_{13}$:(Idle,,get(m$_2$"fuelcheck")/(LO2Chk,-,-)

**T**$_{14}$:(LO2Chk,[LO2leak=true|LO2PressureOk=flase])/(LO2Chk,send(m$_{B4}$))

**T**$_{15}$:(LO2Chk,[LO2leak=false&LO2PressureOK=true])/(HeliumChk,-,-)

**T**$_{16}$:(HeliumChk,[Heluimleak=true|HeliumPressureOK=false])/(HeliumChk,- ,send(m$_{B5}$))

**T**$_{17}$:(HeliumChk,[Heliumleak=false&HeliumPressureOK=true])/(LH2Chk,-,-)

**T**$_{18}$:(LH2Chk,[LH2leak=true|LH2PressureOk!=false])/(LH2Chk,send(m$_{B6}$))

**T**$_{19}$:(LH2Chk,[LH2leak=false&LH2PressureOK=true])/(LH2Chk,,send(m$_{I3}$))

**T**$_{20}$:(Idle,,get(m3"PreFlight")/(INSTChk,-,-)

**T**$_{21}$:(INSTChk,[ChkcksumOK=false|LaunchConductCommOk=false])/(INSTChk,send(m$_{B12}$))

**Table 9** continued

$\mathbf{T}_{22}$:(INSTChk,[ChecksumOk=true&LaunchConductCommOk=true])/(CryoTesting,-,-)

$\mathbf{T}_{23}$:(CryoTesting,[IntTempOK=false|IntPressureOk=false])/(CryoTesting,send($\mathrm{m}_{B13}$))

$\mathbf{T}_{24}$:(CryoTesting,[IntTempOK=true&IntPressureOk=true])/(ChillDown,-,-)

$\mathbf{T}_{25}$:(ChillDown,[IntTempOK=false|InterPssurOK=false])/(ChillDown,,send($\mathrm{m}_{B14}$)

$\mathbf{T}_{26}$:(ChillDown,[IntTemIOK=true&IntPressurOK=true])/(BatteryChk,-,-)

$\mathbf{T}_{27}$:(BatteryChk,[BatBeryPresent=false|PowerLevelOK=false|BatteryLifeOK=false])/(BatteryChk,-,send($\mathrm{m}_{B7}$))

$\mathbf{T}_{28}$:(BatteryChk,[BatteryPresent=true&PowerLevelOK=true&ButteryLifeOK=true])/(InitiateFueling,-,-)

$\mathbf{T}_{29}$:(InitiateFueling,[TankPressureOK=false|FuelLevelOK=false|TankTempOK=false])/(InitiateFueling,send($\mathrm{m}_{B8}$))

$\mathbf{T}_{30}$:(InitiateFueling,[TankPressureOK=true&FuelLevelOK=true&TankTempOK=true])/(InitiateFueling,send($\mathrm{m}_{I4}$)

$\mathbf{T}_{31}$:(Idle,,get($\mathrm{m}_4$”Flight”)/(InternalBattery,-,-)

$\mathbf{T}_{32}$:(InternalBattery,[SwitchToBatteryOK=false|PowerLevelOK=false])/(InternalBattery,send($\mathrm{m}_{B9}$)

$\mathbf{T}_{33}$:(InternalBattery,[SwitchToBatteryOK=true&PowerLevelOK=true])/(FlightCommand,-,-)

$\mathbf{T}_{34}$:(FlightCommand,[StartFlight=true],StartFlight)/(Success,-,send($\mathrm{m}_{I5}$))

$m_{B1}$:$(m_{B1}, e_{B1}, G_1)$

$m_{B2}$:$(m_{B2}, e_{B2}, G_1)$

$m_{B3}$:$(m_{B3}, e_{B3}, G_1)$

$m_{B4}$:$(m_{B4}, e_{B4}, G_2)$

$m_{B5}$:$(m_{B5}, e_{B5}, G_2)$

$m_{B6}$:$(m_{B6}, e_{B6}, G_2)$

$m_{B7}$:$(m_{B7}, e_{B7}, G_3)$

$m_{B8}$:$(m_{B8}, e_{B8}, G_3)$

$m_{B9}$:$(m_{B9}, e_{B9}, G_3)$

$m_{B10}$:$(m_{B10}, e_{B10}, G_4)$

$m_{B11}$:$(m_{B11}, e_{B11}, G_4)$

$m_{B12}$:$(m_{B12}, e_{B12}, G_5)$

$m_{B13}$:$(m_{B13}, e_{B13}, G_5)$

$m_{B14}$:$(m_{B14}, e_{B14}, G_5)$

apply the algorithm in Hessel and Pettersson (2007) to the graph in Fig. 9 by imposing the edge coverage criteria, we obtain the test paths shown in Table 10. Note that this approach forces a proper prioritization if the tests are executed in order, i.e., there is no need for extra test prioritization rules.

The difference between our approach and those that use statecharts such as Sánchez and Felder (2003), Ariss et al. (2011), Kim et al. (2010) is that our approach is used to explicitly model systems (with communication edges) where the behavior process and the failure process intersect. Therefore, paths can be produced. These paths can be used for feasibility testing and planning for mitigation actions, and mitigation testing. It is also possible to manipulate sensor values and create failure events during system testing. Moreover, in our approach, different levels of details can be used for different testing

**Table 10** Aerospace launch system test paths

$\mathbf{r}_1$: Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_5}$ CountDownClockReset]

$\mathbf{r}_2$: ECSInitialization[idle$\xrightarrow{T_8}$AirConditioning$\xrightarrow{T_{10}}$NitrogenPurge$\xrightarrow{T_{11}}$ NitrogenPurge]

$\mathbf{r}_3$: FuelCheck[Idle$\xrightarrow{T_{13}}$LO2Chk$\xrightarrow{T_{15}}$HeliumChk$\xrightarrow{T_{17}}$LH2Chk$\xrightarrow{T_{18}}$LH2Chk]

$\mathbf{r}_4$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk]$\xrightarrow{m_{b12}}$S9$\xrightarrow{m_5}$S11$\xrightarrow{T_{51}}$S12

$\mathbf{r}_5$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$
BatteryChk$\xrightarrow{T_{28}}$InitiaFueling$\xrightarrow{T_{29}}$InitiateFueling]

$\mathbf{r}_6$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$ BatteryChk]$\xrightarrow{m_{b8}}$S5$\xrightarrow{T_{39}}$S6

$\mathbf{r}_7$: Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_2}$NetworkConnection]$\xrightarrow{m_{b1}}$S0 $\xrightarrow{T_{31}}$S1

$\mathbf{r}_8$: Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn]$\xrightarrow{m_{b2}}$S0$\xrightarrow{T_{31}}$ S1

$\mathbf{r}_9$: Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_5}$ CountDownClockReset]$\xrightarrow{m_{b2}}$S0$\xrightarrow{T_{31}}$S1

$\mathbf{r}_{10}$: Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_4}$
HazardLightsOn$\xrightarrow{T_5}$CountDownClock Reset]

$\mathbf{r}_{11}$: Initialization[Idle$\xrightarrow{T_1}$NetworkConnection$\xrightarrow{T_3}$HazardLightsOn$\xrightarrow{T_5}$
CountDownClockReset]$\xrightarrow{m_1}$ECSInitialization[Idle$\xrightarrow{T_8}$AirConditioning$\xrightarrow{T_{10}}$NitrogenPurge]

$\mathbf{r}_{12}$: ECSInitialization[Idle$\xrightarrow{T_8}$AirConditioning$\xrightarrow{T_9}$Airconditioning$\xrightarrow{T_{10}}$ NitrogenPurge]

$\mathbf{r}_{13}$: ECSInitialization[Idle$\xrightarrow{T_8}$AirConditioning$\xrightarrow{T_{10}}$NitrogenPurge]$\xrightarrow{m_{b11}}$S7$\xrightarrow{m_4}$ S11$\xrightarrow{T_{51}}$S12

$\mathbf{r}_{14}$: ECSInitialization[Idle$\xrightarrow{T_8}$AirConditioning$\xrightarrow{T_{10}}$NitrogenPurge]$\xrightarrow{m_2}$FuelChk
[idle$\xrightarrow{T_{13}}$LO2Chk$\xrightarrow{m_{15}}$HeliumChk]$\xrightarrow{m_{b5}}$S3

$\mathbf{r}_{15}$: FuelChk[Idle$\xrightarrow{T_{13}}$LO2Chk$\xrightarrow{T_{14}}$LO2Chk$\xrightarrow{T_{15}}$HeliumChk]$\xrightarrow{m_{B5}}$S4

$\mathbf{r}_{16}$: FuelChk[Idle$\xrightarrow{T_{13}}$LO2Chk$\xrightarrow{T_{15}}$HeliumChk$\xrightarrow{T_{16}}$HeliumChk]$\xrightarrow{m_{B5}}$S4

$\mathbf{r}_{17}$: FuelChk[Idle$\xrightarrow{m_{13}}$LO2Chk$\xrightarrow{T_{15}}$HeliumChk$\xrightarrow{T_{17}}$LH2Chk]$\xrightarrow{m_{I3}}$PreFlight[
Idle$\xrightarrow{T_{20}}$InstrumentChk]$\xrightarrow{m_{B12}}$S9$\xrightarrow{m_5}$S11$\xrightarrow{T_{51}}$S12

$\mathbf{r}_{18}$: PreFlight[Idle$\xrightarrow{T_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting]$\xrightarrow{m_{B13}}$S9$\xrightarrow{m_5}$S11$\xrightarrow{T_{51}}$ S12

$\mathbf{r}_{19}$: PreFlight[Idle$\xrightarrow{T_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown]$\xrightarrow{m_{B14}}$ S9$\xrightarrow{m_5}$S11$\xrightarrow{T_{51}}$S12

$\mathbf{r}_{20}$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{23}}$CryoTesting$\xrightarrow{T_{24}}$
ChillDown$\xrightarrow{T_{26}}$BatteryChk$\xrightarrow{T_{28}}$InitiaFueling]

$\mathbf{r}_{21}$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{25}}$
ChillDown$\xrightarrow{T_{26}}$BatteryChk$\xrightarrow{T_{28}}$InitiaFueling]

**Table 10** continued

| |
|---|
| $\mathbf{r}_{22}$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$ BatteryChk$\xrightarrow{T_{27}}$BatteryChk$\xrightarrow{T_{28}}$InitiaFueling] |
| $\mathbf{r}_{23}$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$ BatteryChk$\xrightarrow{T_{28}}$InitiaFueling]$\xrightarrow{m_{B8}}$S5$\xrightarrow{T_{39}}$S6 |
| $\mathbf{r}_{24}$: PreFlight[Idle$\xrightarrow{m_{20}}$InstrumentChk$\xrightarrow{T_{22}}$CryoTesting$\xrightarrow{T_{24}}$ChillDown$\xrightarrow{T_{26}}$ BatteryChk$\xrightarrow{T_{28}}$InitiaFueling]$\xrightarrow{m_{4}}$Flight[idle$\xrightarrow{T_{31}}$IntBattery$\xrightarrow{T_{33}}$ FlightCommand$\xrightarrow{T_{34}}$Success] |
| $\mathbf{r}_{25}$: Flight[Idle$\xrightarrow{T_{31}}$IntBattery$\xrightarrow{T_{32}}$IntBattery$\xrightarrow{T_{33}}$FlightCommand$\xrightarrow{T_{34}}$Success] |

purposes. For example, if we want to test the system, we can look at every GCEFSM as a unit and not worry about the GCEFSMs' internal details (transitions and states) since we know how they behave.

When we compared the number of states and transitions produced by our integrated approach with those of Sánchez and Felder (2003) on this aerospace launch system example, we found that our ICEFSM contains 41 states and 117 transitions, whereas the EFSM model of Sánchez and Felder (2003) will contain at least 4316 states and 8335 transitions.

## 5 Scalability

### 5.1 Comparison of case studies

We compare the number of nodes and transitions between the model integration approach presented here and Sánchez and Felder (2003) approach (EFSM from statecharts and FTs). First, we compare the number of nodes of three case studies:

1. The railroad crossing control system (RCCS) in Gario and Andrews (2014),
2. The gas burner example (GB) of Gario (2014), and
3. The launch vehicle (LV) presented here.

Table 11 shows the comparison of case studies that we have actually conducted. The left column identifies the case study. The column labeled BM reports the number of states (S) and transitions (T) in the behavioral model, respectively. The column labeled FM reports the number of leaves in the fault tree and how many gates of various types are in the FT. The columns marked CEFSMs and EFSMs report the number of states and transitions in our approach versus Sánchez and Felder (2003) approach. Note that our approach roughly increases states and transitions as a proportion of the number of leaves in the fault tree, while Sánchez et al.'s approach shows an exponential increase. Clearly, our approach looks more scalable. To investigate this further, we used our tool as a simulator with a range of model and fault tree sizes.

### 5.2 Simulation with increasing size

We developed a tool to calculate the number of states and transitions of the integrated behavioral and fault models according to the approach's transformation rules (CEFSMs

**Table 11** Comparison

| System | BM | | FM | | | | | CEFSMs | | EFSMs | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | S | T | leaves | AND | OR | XOR | Timing | S | T | S | T |
| GB | 13 | 15 | 5 | 3 | 1 | 0 | 2 | 27 | 55 | 79 | 162 |
| RRC | 14 | 19 | 8 | 2 | 5 | 0 | 0 | 28 | 70 | 303 | 514 |
| LV | 21 | 39 | 14 | 0 | 10 | 0 | 0 | 41 | 117 | 4316 | 8335 |

*S* state, *T* transition, timing = timing gates

with FTs) and to estimate the number of nodes and transitions according to Sánchez and Felder (2003) approach (EFSM from statecharts and FTs).

## 5.3 In CEFSM

The tool calculates the number of nodes and transitions of the integrated model by adding the number of the nodes and transitions of the behavioral model to the number of the states and transitions of the GCEFSM part of the model. As we have seen in section 3, the FT gates are transformed into a collection of GCEFSMs. Every GCEFSM has a specific number of nodes and transitions. Thus, the tool calculates the number of nodes and transitions of the ICEFSM based on the number and type of gates.

## 5.4 For EFSM in Sánchez and Felder (2003)

The tool estimates the number of nodes and transitions of the integrated model according to the approach's transformation rules. The integration rules of the Sánchez and Felder (2003) approach uses the minimum cut set of the leave node events. For every member of the cut set they create an independent region, add a state to the behavioral model, or do nothing. This depends on whether the event already existed in the behavioral model, or may need human intervention to decide whether to model the cut as an independent region or to add it to the behavioral model as a single state and transition. Therefore, we calculated the size of the integrated model based on these options repeatedly and computed an average. Each time, we change the percentage of creating an independent region. We run the tool 10 times for each input data varying the probability of creating an independent region between 50 and 60 %. Then, we calculated the average of these runs.

We fed the tool with input data of different size ranges of BM and FM. The behavioral models vary from 13 states and 15 transitions to 50 states and 60 transitions while the fault trees vary from five leaves to 19 leaves as shown in Table 12. The fault tree is constructed of leaves which denote the number of input events to the fault tree and different types and numbers of gates, AND (0–6) gates, OR (1–10) gates, XOR (0–6) gates, and Timing gates (0–4) gates. The AND gate includes AND gate, Priority AND gate, and Inhibit gate.

We assume that the behavioral model is connected, and no part of is isolated; therefore, the number of transitions must not be less than the number of states $-1$. We also assumed that the fault tree is a binary tree where the number of gates equals the number of leaf nodes $-1$. The timing gates, however, are excluded from this calculation because they take only one event and they appear only at the leaf nodes. However, we need to consider them to calculate the number of states and transitions of our integrated model, the ICEFSM. We assume that every behavioral model is integrated with every fault model. Therefore, the

**Table 12** Simulation data and results

| BM | | FM | | | | | CEFSMs | | EFSMs | |
|---|---|---|---|---|---|---|---|---|---|---|
| S | T | Leaves | AND | OR | XOR | Timing | S | T | Avg (S) | Avg (T) |
| 13 | 15 | 5 | 3 | 1 | 0 | 2 | 27 | 55 | 79 | 162 |
| 13 | 15 | 7 | 3 | 2 | 1 | 1 | 28 | 64 | 178 | 304 |
| 13 | 15 | 8 | 2 | 5 | 0 | 0 | 27 | 66 | 263 | 416 |
| 13 | 15 | 14 | 0 | 10 | 0 | 0 | 33 | 93 | 2672 | 3280 |
| 13 | 15 | 19 | 6 | 5 | 6 | 4 | 59 | 158 | 18264 | 21342 |
| 13 | 17 | 5 | 3 | 1 | 0 | 2 | 27 | 57 | 79 | 174 |
| 13 | 17 | 7 | 3 | 2 | 1 | 1 | 28 | 66 | 178 | 332 |
| 13 | 17 | 8 | 2 | 5 | 0 | 0 | 27 | 68 | 263 | 456 |
| 13 | 17 | 14 | 0 | 10 | 0 | 0 | 33 | 95 | 2672 | 3691 |
| 13 | 17 | 19 | 6 | 5 | 6 | 4 | 59 | 160 | 18264 | 24152 |
| 15 | 19 | 5 | 3 | 1 | 0 | 2 | 29 | 59 | 92 | 197 |
| 15 | 19 | 7 | 3 | 2 | 1 | 1 | 30 | 68 | 206 | 374 |
| 15 | 19 | 8 | 2 | 5 | 0 | 0 | 29 | 70 | 303 | 514 |
| 15 | 19 | 14 | 0 | 10 | 0 | 0 | 35 | 97 | 3083 | 41335 |
| 15 | 19 | 19 | 6 | 5 | 6 | 4 | 61 | 162 | 21074 | 27003 |
| 17 | 18 | 5 | 3 | 1 | 0 | 2 | 31 | 58 | 104 | 202 |
| 17 | 18 | 7 | 3 | 2 | 1 | 1 | 32 | 67 | 233 | 376 |
| 17 | 18 | 8 | 2 | 5 | 0 | 0 | 31 | 69 | 343 | 5115 |
| 17 | 18 | 14 | 0 | 10 | 0 | 0 | 37 | 96 | 3494 | 3958 |
| 17 | 18 | 19 | 6 | 5 | 6 | 4 | 63 | 161 | 23883 | 25640 |
| 19 | 19 | 5 | 3 | 1 | 0 | 2 | 33 | 59 | 116 | 219 |
| 19 | 19 | 7 | 3 | 2 | 1 | 1 | 34 | 68 | 260 | 405 |
| 19 | 19 | 8 | 2 | 5 | 0 | 0 | 33 | 70 | 384 | 549 |
| 19 | 19 | 14 | 0 | 10 | 0 | 0 | 39 | 97 | 3905 | 4194 |
| 19 | 19 | 19 | 6 | 5 | 6 | 4 | 65 | 162 | 26693 | 27086 |
| 21 | 39 | 5 | 3 | 1 | 0 | 2 | 35 | 79 | 128 | 352 |
| 21 | 39 | 7 | 3 | 2 | 1 | 1 | 36 | 88 | 288 | 694 |
| 21 | 39 | 8 | 2 | 5 | 0 | 0 | 35 | 90 | 424 | 971 |
| 21 | 39 | 14 | 0 | 10 | 0 | 0 | 41 | 117 | 4316 | 8335 |
| 21 | 39 | 19 | 6 | 5 | 6 | 4 | 67 | 182 | 29503 | 55225 |
| 30 | 40 | 5 | 3 | 1 | 0 | 2 | 44 | 80 | 183 | 408 |
| 30 | 40 | 7 | 3 | 2 | 1 | 1 | 45 | 89 | 411 | 777 |
| 30 | 40 | 8 | 2 | 5 | 0 | 0 | 44 | 91 | 606 | 1069 |
| 30 | 40 | 14 | 0 | 10 | 0 | 0 | 50 | 118 | 6165 | 8677 |
| 30 | 40 | 19 | 6 | 5 | 6 | 4 | 76 | 183 | 42147 | 56817 |
| 40 | 45 | 5 | 3 | 1 | 0 | 2 | 54 | 85 | 244 | 493 |
| 40 | 45 | 7 | 3 | 2 | 1 | 1 | 55 | 94 | 548 | 921 |
| 40 | 45 | 8 | 2 | 5 | 0 | 0 | 54 | 96 | 808 | 1257 |
| 40 | 45 | 14 | 0 | 10 | 0 | 0 | 60 | 123 | 8220 | 9858 |
| 40 | 45 | 19 | 6 | 5 | 6 | 4 | 86 | 188 | 56196 | 64049 |
| 50 | 60 | 5 | 3 | 1 | 0 | 2 | 64 | 100 | 305 | 639 |
| 50 | 60 | 7 | 3 | 2 | 1 | 1 | 65 | 109 | 685 | 1204 |

**Table 12** continued

| BM | | FM | | | | | CEFSMs | | EFSMs | |
|---|---|---|---|---|---|---|---|---|---|---|
| S | T | Leaves | AND | OR | XOR | Timing | S | T | Avg (S) | Avg (T) |
| 50 | 60 | 8 | 2 | 5 | 0 | 0 | 64 | 111 | 1010 | 1648 |
| 50 | 60 | 14 | 0 | 10 | 0 | 0 | 70 | 138 | 10275 | 13093 |
| 50 | 60 | 19 | 6 | 5 | 6 | 4 | 96 | 203 | 70245 | 85330 |

*S* state, *T* transition, timing= timing gates, Avg (*S*) = the average of the number of states, and Avg (*T*) the average of the number of transitions of 10 runs

simulator calculates the size (states and transitions) of the integrated model of every behavioral model with every fault model as inputs.

We started with the relatively small behavioral model (GB) with 13 states and 15 transitions. This model is integrated with different fault trees as shown in Table 12. We can see that the number of states and transitions of the integrated model of the EFSM approach grows exponentially. The number of the states produced by our integration approach grows from 21 to 59, and the number of transitions grows from 41 to 137, whereas the number of states produced by the EFSM integration approach grows on average from 79 to 70,245 and transitions from 162 to 85,330. It is very clear that the numbers of the states and transitions of both approaches are quite different.

Table 12 shows that even for the larger BMs and larger fault trees with more leaves, our approach produces integrated models of efficient sizes, while the approach by Sánchez *et al.* very quickly reaches scalability limits. Figures 36, 37, 38, and 39 show the growth of the integrated models as a function of the number of leaves in the fault tree. While Sánchez *et al.*'s approach is highly affected by the number of leaves in the fault tree, our approach is not.

Figure 39 represents the (50 states and 60 transitions) model. We can clearly notice that the trend of the curves in Figs. 36 and 39 is the same. The only difference is the number of states and transitions which depends on the size of the behavioral and the fault models.

## 5.5 Scalability of CEFSM approach

Our approach used CEFSMs to integrate the fault model with the behavioral model. For small behavioral models, integrating the FTs makes a large difference in the integrated model because of the FT's relatively large size compared to the size of the behavioral model. Although the growth of integrating small behavioral models with FTs seems relatively large compared to the behavioral model, it does not mean that the integrated model is really large. In fact, it may be even smaller than other large behavioral models. However, for large behavioral models, the increase in the size of the integrated model is relatively small when many or a large FT is integrated.

## 6 Conclusion and future work

This paper describes a testing technique that allows testing of safety-critical systems and evaluating its scalability. We defined a test generation framework that takes a behavioral and a fault model, transforms the fault model, and integrates it with the behavioral model. The integrated model can then be used with a variety of test criteria to generate tests. We

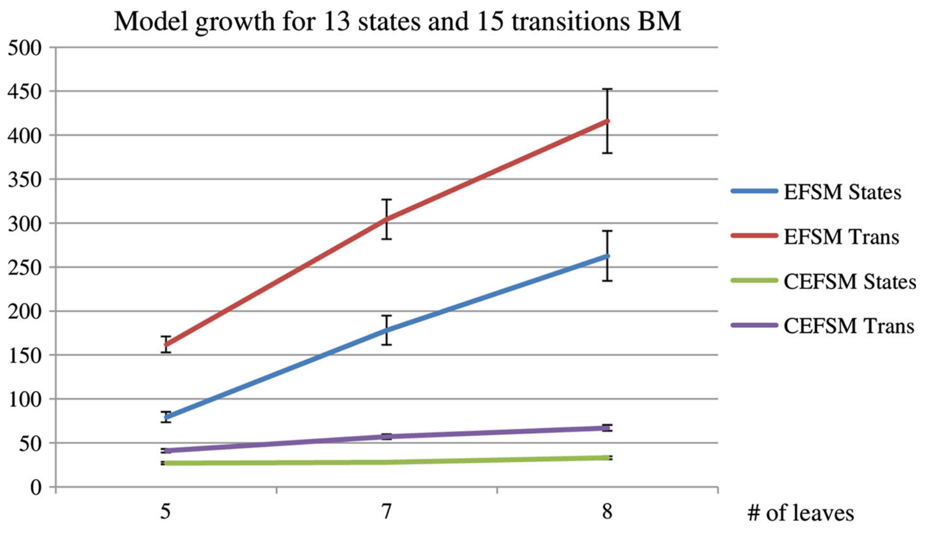


**Fig. 36** EFSM and CEFSM approaches model growth for 13 S and 15 T behavioral model

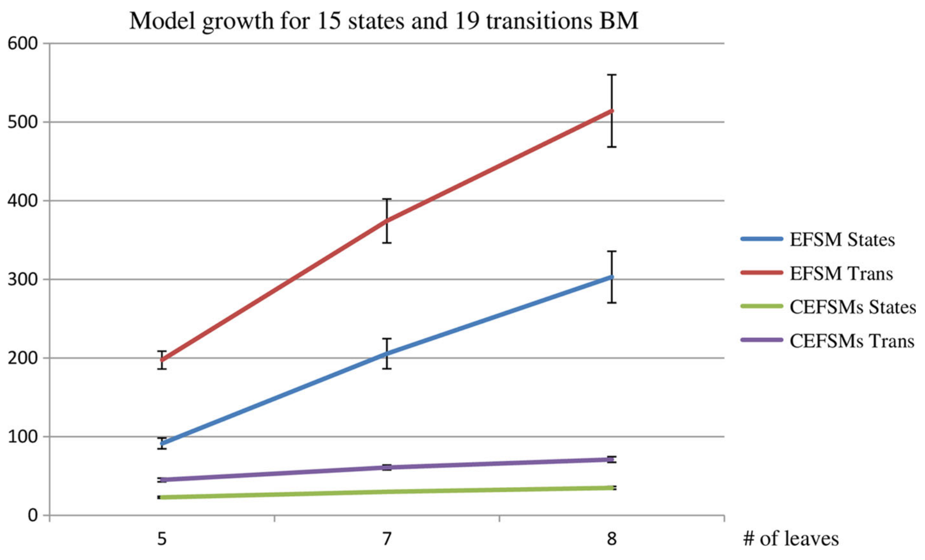


**Fig. 37** EFSM and CEFSM approaches model growth for 15 S and 19 T behavioral model

provided a specific set of transformation and integration rules for a CEFSM as a behavioral model and a fault tree as a fault model. The key to this integration was compatibility in naming events and conditions for event occurrence. While one might be tempted to skip FTA and include the fault information “ad hoc” in the CEFSM directly, this is unsystematic and error-prone. It also fails to provide a proper FTA, an important part of developing safety-critical systems. Model scalability is also investigated in this paper. To this end, we developed a tool that estimates the number of states and transitions both for

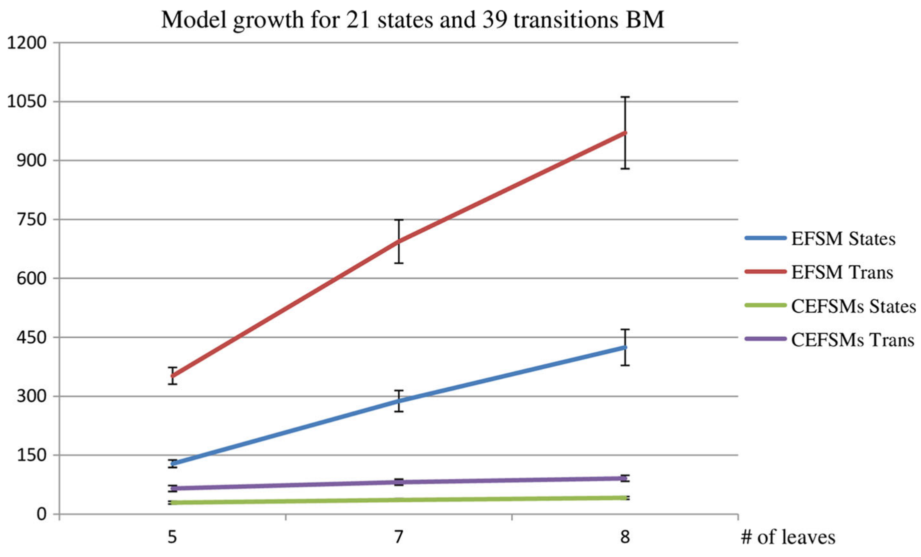


**Fig. 38** EFSM and CEFSM approaches model growth for 21 S and 39 T behavioral model

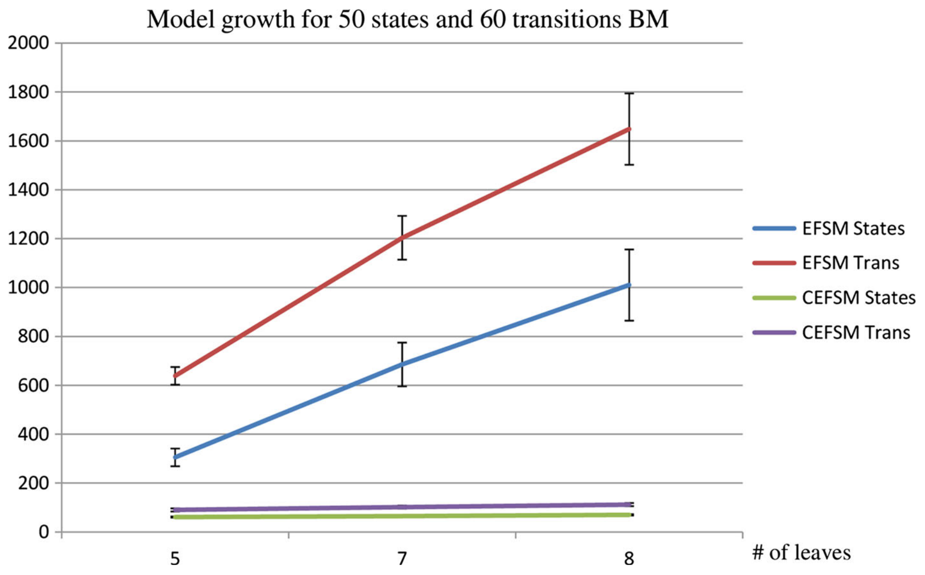


**Fig. 39** EFSM and CEFSM approaches model growth for 50 S and 60 T behavioral model

our approach and Sánchez and Felder (2003) approach. We then compared model sizes and investigated scalability. We clearly showed that our approach is more scalable.

Future work includes providing a testing approach for testing proper mitigation and a selective regression testing approach when there are changes to the behavioral model or the fault model.

**Acknowledgments** This work was partially supported by NSF Grant Numbers 0934413, 1127947, and 1332078.

## Appendix

### INHIBIT gate

INHIBIT is similar to the AND gate. They have the same states and transitions. The only difference is that the predicate for the transitions $T_2$ and $T_3$ should include the enabling condition. We do not need to have a separate gate representation for NOT gate since we can express it in any predicate. If we want to negate any event, we can use the NOT logical operator inside the gate that the negated event is one of its inputs.

### XOR gate

This gate is slightly different from the AND gate, although it has the same structure and same number of transitions and states. At this gate, it is necessary to distinguish between the event that has not occurred in the first place and the one whose status is false. The representation of GCEFSM XOR gate is shown in Fig. 40. $T_0$ to $T_3$ are the possible transitions that may be taken based on their predicates.

$T_0 : (S_0, [NoOfOccurredEvents = 0\&e_j.eOccurrence = true], get(m_j))/(S_0,\ update(events))$
$T_1 : (S_0, [NoOfOccurredEvents = 1\&e_j.eOccurrence = true\&xor(events) = false], get(m_j))/(S_0, update(events), -)$
$T_2 : (S_0, [NoOfOccurredEvents = 1\&e_j.eOccurrence = true\&xor(events) = true], get(m_j))/(S_1, update(events), Send(gateOccurred))$
$T_3 : (S_1, [inputStatusChanged(e_j) = true], get(m_j))/(S_0, update(events), Send(gatenotoccurred))$

### Timing an event gate

FT gates such as AND, OR, and INHIBIT are well defined and can be syntactically represented. Events in FT can be simple or composed. A composed event can be decomposed further to simple events or a timed simple event. A timed simple event is the simple event that should occur for a specific period of time to contribute to a hazard. However, FT has no timing gates. Therefore, we need to have a representation that can handle the timing issue (either a minimum or maximum timing). A CEFSM can be supplemented with timers and timer-related operations. A timer is set with a time value during a transition. If the timer is not canceled by the CEFSM, the timer will generate a time expiration signal after the time period has been exceeded (Byun et al. 2001, 2002; Byun 2003; Byun and Sanders 2005, 2006). Thus, we introduce this gate that can time an event and the gate in the subsection 7 that deals with the timing intervals. This gate works as follows. Upon receiving a message that indicates the occurrence of the event, the transition

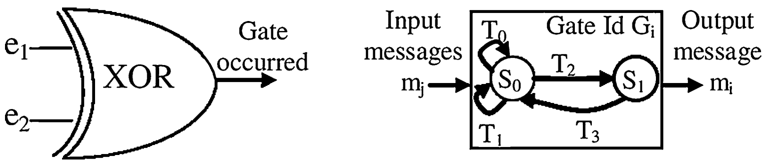


**Fig. 40** XOR gate representation in FT and GCEFSM

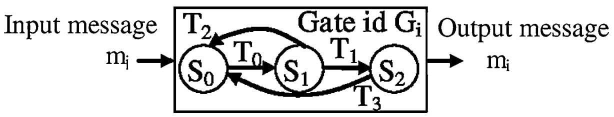


**Fig. 41** Event timer GCEFSM

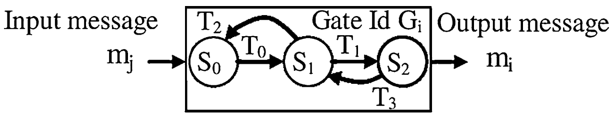


**Fig. 42** Timing continuous intervals GCEFSM

$T_0$ takes place which starts the timer. When the time expires and no further "gate not occurred" message was received that indicates that the event is no longer happening, the transition $T_2$ is taken and sends a "gate occurred" message. Otherwise, the gate does not occur. $T_2$ is taken when the event status $e_i$.eStatus changes to false.

$$T_0 : (S_0, [e_j.eStatus = true], get(m_j))/(S_1, setTimer(v, Timer_i), -)$$
$$T_1 : (S_1, time - out)/(S_2, -, Send(GateOccurred))$$
$$T_2 : (S_1, e_j.eStatus = false], get(m_j))/(S_0, reset(Timer_i); update(events))$$
$$T_3 : (S_2, e_i.eStatus = false], get(m_i))/(S_0, reset(Timer_i); update(events), Send(GateNotOccurred))$$

*Timing an event for continuous intervals gate*

Some event may need to be timed for continuous intervals. For example, we may need to observe an occurrence of an event every consecutive 5 sec as long as the system is operational. Fig. 42 shows that as long as the transition $T_0$ is fired and $T_2$ was not, the event will be timed for fixed consecutive amounts of time and it keeps timing until the status of the event *$e_i$.eStatus* changes to false. Upon receiving this event change, the transition $T_3$ to the state $S_1$ is fired sending out a "gate not occurred" message.

$$T_0 : (S_0, [e_j.eStatus = true], get(m_j))/(S_1, setTimer(v, Timer_i), Send(GateOccurred))$$
$$T_1 : (S_1, time - out)/(S_2, -, Send(GatenotOccurred))$$
$$T_2 : (S_1, [e_j.eStatus = false], get(m_j))/(S_0, reset(Timer_i), Send(GateNotOccurred))$$
$$T_3 : (S_2, setTimer(v, Timer_i))/(S_1, -, Send(GateOccurred))$$

## References

Amberkar, S., Murray, M. T., Demerly, J. D., D'Ambrosio, J. G., & Czerny, B. J. (2001). A comprehensive hazard analysis technique for safety-critical automotive systems.

Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., & Sabina, S. (2009). Automatic test generation for coverage analysis of ERTMS software. In *International Conference on Software Testing Verification and Validation, 2009. ICST '09* (pp. 303–306). Washington, DC, USA.

Bobbio, A., Portinale, L., Minichino, M., & Ciancamerla, E. (2001). Improving the analysis of dependable systems by mapping fault trees into bayesian networks. *Reliability Engineering and System Safety, 71*(3), 249–260.

Boroday, S., Petrenko, A., Groz, R., & Quemener, Y. M. (2002). Test generation for CEFSM combining specification and fault coverage. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02 (pp. 355–372). Deventer: Kluwer, B.V.

Boudali, H., & Dugan, J. B. (2005). A discrete-time bayesian network reliability modeling and analysis framework. *Reliability Engineering and System Safety*, *87*, 337–349.

Bourhfir, C., Aboulhamid, E., Dssouli, R., & Rico, N. (2001). A test case generation approach for conformance testing of SDL systems. *Computer Communications*, *24*(3–4), 319–333.

Bourhfir, C., Dssouli, R., Aboulhamid, E. M., & Rico, N. (1998). A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols. In *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, IWTCS (pp. 275–290). Deventer: Kluwer, B.V.

Bourhfir, C., Dssouli, R., Aboulhamid, M., & Rico, N. (1999). A test case generation tool for conformance testing of SDL systems. In *SDL forum* (pp. 405–420).

Brand, D., & Zafiropulo, P. (1983). On communicating finite-state machines. *Journal of ACM*, *30*(2), 323–342.

Buchacker, K., & Friedrich Alexander Universitht, I. (1999). Combining fault trees and petri nets to model safety-critical systems. In *Society for Computer Simulation International* (pp. 439–444).

Byun, Y. (2003). *Pattern-based design and validation of communication protocols*. Ph.D. thesis, University of Florida, Gainesville, FL, USA.

Byun, Y., Beverly, S., & Chung, K. (2002). A pattern language for communication protocols. In *Proceedings of the 9th Conference on Pattern Languages of Programs (PLoP).*

Byun, Y., & Sanders, B. A. (2005). A pattern-based development methodology for communication protocols. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, & Roger L. Wainwright, (Eds.) *SAC* (pp. 1524–1528). ACM.

Byun, Y., & Sanders, B. A. (2006). A pattern-based development methodology for communication protocols. *Journal of Information Science and Engineering*, *22*(2), 315–335.

Byun, Y., Sanders, B. A., & Keum, C. (2001). Design patterns of communicating extended finite state machines in sdl. In *In proceedings of the 8th Conference on Pattern Languages if Programs.*

Cheng, K. T., & Krishnakumar, K. S. (1993). Automatic functional test generation using the extended finite state machine model. In *30th Conference on Design Automation* (pp. 86–91).

Czerny, B. J., Ambrosio, J. G., Murray, B. T., & Sundaram, P. (2005) Effective application of software safety techniques for automotive embedded control systems. *Engineering, 1*(724).

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G.C., et al. (1999). Model-based testing in practice. In *ICSE* (pp. 285–294).

Ek, A., Grabowski, J., Hogrefe, D., Jerome, R., Koch, B., & Schmitt II, M. (1997). Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. In *SDL forum* (pp. 245–260).

El Ariss, O., Xu, D., & Wong, W. E. (2011). Integrating safety analysis with functional modeling. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, *41*(4), 610–624.

Ericson, C. A. (2005). *Hazard analysis techniques for system safety*. New Jersey: wiley-interscience.

Flammini, F., Marrone, S., Iacono, M., Mazzocca, N., & Vittorini, V. (2014). A multiformalism modular approach to ERTMS/ETCS failure modeling. *International Journal of Reliability, Quality and Safety Engineering*, *21*(01), 1–29.

Flammini, F., Mazzocca, N., Iacono, M., & Marrone, S. (2005). Using repairable fault trees for the evaluation of design choices for critical repairable systems. In *Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems Engineering, HASE '05* (pp. 163–172). Washington, DC, USA, 2005. IEEE Computer Society.

France, R., & Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *2007 future of software engineering, FOSE '07* (pp. 37–54). Washington, DC: IEEE Computer Society.

Garavel, H., Helmstetter, C., Ponsini, O., & Serwe, W. (2009). Verification of an industrial systemC/TLM model using LOTOS and CADP. In *MEMOCODE* (pp. 46–55).

Garavel, H., Lang, F., Mateescu, R., & Serwe, W. (2013). CADP 2011: a toolbox for the construction and analysis of distributed processes. *The International Journal on Software Tools for Technology Transfer (STTT)*, *15*(2), 89–107.

Garavel, H., Mateescu, R., & Serwe, W. (2013). Large-scale distributed verification using CADP: Beyond clusters to grids. *Electronic Notes Theory Computer Science*, *296*, 145–161.

Gario, A. (2014). *Fail-Safe testing of safety-critical systems*. PhD thesis, University of Denver, Denver, CO, USA, 11.

Gario, A., & Andrews, A. (2014). Fail-safe testing of safety-critical systems. In *Software Engineering Conference (ASWEC)*, 2014 23rd Australian (pp. 190–199). IEEE.

Gario, A., Andrews, A., & Hagerman, S. (2014). Testing of safety-critical systems: An aerospace launch application. In *Aerospace Conference, 2014 IEEE* (pp. 1–17). IEEE.

Ghazel, M. (2014). Formalizing a subset of ERTMS/ETCS specifications for verification purposes. *Transportation Research Part C: Emerging Technologies*, *42*, 60–75.

Di Giorgio, A., & Liberati, F. (2011). Interdependency modeling and analysis of critical infrastructures based on dynamic bayesian networks. In *19th Mediterranean Conference on Control Automation (MED)*, 2011 (pp. 791–797).

Henniger, O., Lu, M., & Ural, H. (2004). Automatic generation of test purposes for testing distributed systems. In Alexandre Petrenko & Andreas Ulrich (Eds.), *Formal approaches to software testing* (Vol. 2931, pp. 1105–1105). Lecture Notes in Computer Science Berlin/Heidelberg: Springer.

Hessel, A., & Pettersson, P. (2007). A global algorithm for model-based test suite generation. *Electronic Notes in Theoretical Computer Science*, *190*(2), 47–59.

Kaiser, B. (2003). A fault-tree semantics to model software-controlled systems. *Softwaretechnik-Trends*, *23*(3), 33–39.

Kaiser, B. (2005). Extending the expressive power of fault trees. In *Proceedings on Reliability and Maintainability Symposium*, 2005 (pp. 468–474). Alexandria, Virginia.

Kaiser, B., Gramlich, C., & Förster, M. (2007). State/event fault trees—A safety analysis model for software-controlled systems. *Reliability Engineering and System Safety*, *92*(11), 1521–1537.

Kaiser, B., Liggesmeyer, P., & Mäckel, O. (2003). A new component concept for fault trees. In *Proceedings of the 8th Australian workshop on Safety critical systems and software, volume 33 of SCS '03* (pp. 37–46). Darlinghurst: Australian Computer Society Inc.

Keller, R. M. (1976). Formal verification of parallel programs. *Communications of the ACM*, *19*(7), 371–384.

Kim, H., Wong, W. E., Debroy, V., & Bae, D. (2010). Bridging the gap between fault trees and UML state machine diagrams for safety analysis. In *17th Asia Pacific Software Engineering Conference (APSEC)* (pp. 196–205).

Kloos, J., Hussain, T., & Eschbach, R. (2011). Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW 2011)* (pp. 26–33). Los Alamitos, CA: IEEE Computer Society.

Kovács, G., Pap, Z., & Csopaki, G. (2002). Automatic test selection based on CEFSM specifications. *Acta Cybernet*, *15*(4), 583–599.

Leaphart, E. G., Czerny, B. J., Ambrosio, J. G. D., Denlinger, C. L., & Littlejohn, D. (2005). Survey of software failsafe techniques for safety-critical automotive applications. *Engineering, 1*(724).

Lee, D., & Yannakakis, M. (1996). Principles and methods of testing finite state machines- a survey. *Proceedings of the IEEE*, *84*(8), 1090–1123.

Leveson, N. G., & Harvey, P. R. (1983). Analyzing software safety. *IEEE Transactions on Software Engineering*, *SE–9*(5), 569–579.

Li, J. J., & Wong, W. E. (2002). Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC 2002) Proceedings (pp. 181–185).

Marrone, S., Flammini, F., Mazzocca, N., Nardone, R., Vittorini, V. (2014). Towards model-driven V&V assessment of railway control systems. *International Journal on Software Tools for Technology Transfer* (pp. 669–683).

Medikonda, B. S., Ramaiah, P. S., & Gokhale, A. A. (2011). FMEA and fault tree based software safety analysis of a railroad crossing critical system. *Global Journal of Computer Science and Technology GJCST*, *11*, 57–62.

Montani, S., Portinale, L., Bobbio, A., & Codetta-Raiteri, D. (2008). Radyban: A tool for reliability analysis of dynamic fault trees through conversion into dynamic bayesian networks. *Reliability Engineering and System Safety*, *93*(7), 922–932.

Nazier, R., & Bauer, T. (2012). Automated risk-based testing by integrating safety analysis information into system behavior models. In *IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 213–218).

Ortmeier, F., Güdemann, M., & Wolfgang, R. (2007). Formal failure models. In *Proceedings of the 1st IFAC Workshop on Dependable Control of Discrete Systems (DCDS 07)*. Elsevier.

Petricic, A., Crnkovic, I., & Zagar, M. (2008). Models transformation between UML and a domain specific language. In *Eight Conference on Software Engineering Research and Practice in Sweden (SERPS 08)*.

Petricic, A., Lednicki, L., & Crnkovic, I. (2009). Using UML for domain-specific component models. In *Proceedings of the 14th International Workshop on Component-Oriented Programming*.

Raiteri, D. C., Franceschinis, G., Iacono, M., & Vittorini, V. (2004). Repairable fault tree for the automatic evaluation of repair policies. In *2004 International Conference on Dependable Systems and Networks* (pp. 659–668).

Sánchez, M., & Felder, M. (2003). A systematic approach to generate test cases based on faults. In *Argentine Symposium in Software Engineering*, Buenos Aires, Argentina.

Savage, P., Walters, S., & Stephenson, M. (1997). Automated test methodology for operational flight programs. In *Aerospace Conference, 1997. Proceedings, IEEE* (Vol. 4, pp. 293–304).

Sinha, A., & Smidts, C. (2006). An experimental evaluation of a higher-ordered-typed-functional specification-based test-generation technique. *Empirical Software Engineering*, *11*(2), 173–202.

Teradyne Software and Systems Test, (1999). *Testmaster: User's guide*. New Hampshire: Empirix Inc.

Tretmans, J. (2008). Model based testing with labeled transition systems. In *Formal methods and testing* (pp. 1–38).

Tribble, A. C., & Miller, S. P. (2004). Software intensive systems safety analysis. *IEEE Aerospace and Electronic Systems Magazine*, *19*(10), 21–26.

Utting, M., & Legeard, B. (2007). *Practical model-based testing: A tools approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

VASY. CADP (Caesar/Aldebaran Development Package). http://cadp.inria.fr/

Vesely, W., Dugan, J., Fragola, J., Minarick, & Railsback, J. (2002). *Fault tree handbook with aerospace applications*. Washington, DC: Handbook, National Aeronautics and Space Administration.

Vaos, J. M., & McGraw, G. (1998). *Software fault injection: Inoculating programs against errors*. New Jersey: Wiley Computer Pub.

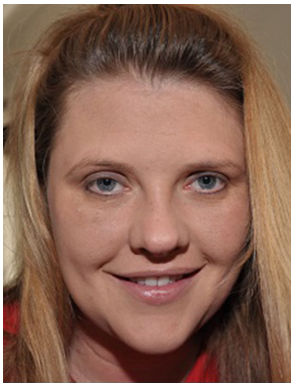
Wada, H., Suzuki, J., & Takada, S. (2005). A model transformation framework for domain specific languages: An approach using UML and attribute-Oriented programming. In *In Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*.

Wang, D., & Pan, J. (2010). An optimization to automatic fault tree analysis and failure mode and effect analysis approaches for processes. In *2010 International Conference on Computer Design and Applications (ICCDA)* (Vol. 3, pp. 153–157).

Xiang, J., Futatsugi, K., & He, Y. (2004). Formal fault tree construction and system safety analysis. In *IASTED Conference on Software Engineering* (pp. 378–384).

**Ahmed Gario** received his BSc, MSc, and PhD degrees in computer science from Tripoli University, Tripoli, Libya; Concordia University, Montreal, Canada; and University of Denver, Denver, CO, USA, respectively. His current research interests include software safety and software testing.

**Anneliese Andrews** is Professor of Computer Science at the University of Denver. Before joining the University of Denver, she held the Huie Rogers Endowed Chair in Software Engineering and served as Associate Director of the School of Electrical Engineering and Computer Science at Washington State University. Dr. Andrews is the author of a text book and over 200 articles in the area of Software and Systems Engineering, particularly software testing, system quality, and reliability. Dr. Andrews holds an MS and PhD from Duke University and a Dipl.-Inf. from the Technical University of Karlsruhe. She served as Editor in Chief of the IEEE Transactions on Software Engineering. She has also served on several other editorial boards including the IEEE Transactions on Reliability, the Empirical Software Engineering Journal, the Software Quality Journal, the Journal of Information Science and Technology, and the Journal of Software Maintenance. Dr. Andrews is currently the DU site Director of a NSF Industry/University Collaborative Research Center for Safety, Security, Search, and Rescue.



**Seana Hagerman** is a PhD candidate at the University of Denver. The focus of her research is on security testing autonomous systems using model-based test methodologies. She has 15 years of experience in software engineering in the aerospace industry. She holds a BS in Computer Science from Colorado Mesa University and an MS in Computer Science from the University of Denver.