CHALMERS
UNIVERSITY OF TECHNOLOGY

Supervisory Control Theory in System Safety Analysis

Supervisory Control Theory in System Safety Analysis

Yuvaraj Selvaraj[1,2(✉)], Zhennan Fei[1], and Martin Fabian[2]

[1] Zenuity AB, Gothenburg, Sweden
{yuvaraj.selvaraj,zhennan.fei}@zenuity.com
[2] Chalmers University of Technology, Gothenburg, Sweden
fabian@chalmers.se

Abstract. Development of safety critical systems requires a risk management strategy to identify and analyse hazards, and apply necessary actions to eliminate or control them as malfunctions could be catastrophic. Fault Tree Analysis (FTA) is one of the most widely used methods for safety analysis in industrial use. However, the standard FTA is manual, informal, and limited to static analysis of systems. In this paper, we present preliminary results from a model-based approach to address these limitations using Supervisory Control Theory. Taking an example from the Fault Tree Handbook, we present a systematic approach to incrementally obtain formal models from a fault tree and verify them in the tool Supremica. We present a method to calculate minimal cut sets using our approach. These compositional techniques could potentially be very beneficial in the safety analysis of highly complex safety critical systems, where several components interact to solve different tasks.

Keywords: Fault tree analysis · Supervisory control theory · Formal methods · System safety · Autonomous driving

1 Introduction

Software development in safety critical systems necessitates a risk management strategy to identify and analyse risks, and to apply the necessary actions to eliminate or control them. The objective of safety analyses, performed during various development phases, is to ensure that the risk of safety violations due to the occurrence of different faults is sufficiently low.

Fault Tree Analysis, FTA [16], is one of the most common methods for safety analysis in various industries. While standard fault trees are simple and informative, they are not free from limitations [3]. Standard FTA is primarily a manual process based on an informal model, i.e., the process relies on the system analysts and domain experts to systematically think about all risks and their possible causes. The lack of formal semantics makes it difficult to verify the correctness of

Supported by FFI, VINNOVA under grant number 2017-05519, *Automatically Assessing Correctness of Autonomous Vehicles–Auto-CAV*.

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-55583-2_1

the safety analysis, especially for rapidly evolving industries like the autonomous driving industry where new edge cases are continuously identified. In complex industrial software controlled systems, safety models must capture many possible interactions between system components, where different interleavings of failure events can either result in a failure or operational state. Standard fault trees are not suitable for modelling temporal, sequential and state dependencies of events. Another notable shortcoming with standard FTA for large and complex systems is the need for safety analyses to be intuitive and compositional. This is crucial in projects where the system of interest comprises interacting sub-systems, possibly delivered by different teams or suppliers.

Though several limitations exist, FTA is one of the widely used safety analysis methods. Different extensions to standard fault trees [10] have been proposed to address some of the limitations. Research on using formal logic in FTA [2,15,17] address the limitation of informal and manual FTA process. Extensions like dynamic fault trees [1], state-event fault trees [4], and temporal fault trees [8] address inability of standard fault trees to model dynamic behaviour. The most widely used extension to include temporal sequence information is dynamic fault trees [1,10]. Over the years, research on the development of model-based dependability analysis (MBDA) [12] techniques have enabled automated dependability analysis. In [12], such emerging MBDA techniques are classified into two paradigms. The first paradigm, termed failure logic synthesis and analysis focuses on automatic construction of failure analyses and the second paradigm, termed behavioural fault simulation focuses on formal verification based techniques. Despite this research, challenges remain in addressing the limitations with standard fault trees and safety analysis [10,12]. Thus any progress in addressing these limitations is helpful. The preliminary results presented in this paper is part of an ongoing endeavour to address the aforementioned limitations by a model-based approach based on Supervisory Control Theory (SCT) [9].

The formal models used in the SCT framework can describe dynamic behaviour, which is often needed to analyse modern and complex safety critical systems. The compositional abstraction based algorithms used in SCT allow automated synthesis and verification of safety models for large and complex systems. These features of the SCT framework makes it possible to define a complete model-based safety analysis approach with automated analysis. To ensure sufficient detail of explanation and some degree of familiarity, we do not present a complex example in this paper; instead we describe our approach using a rather simple example from the *Fault Tree Handbook* [16].

We make three main contributions in this paper. First, we address the issue of informal description of standard fault tree analysis by presenting a systematic approach to incrementally obtain formal models from a fault tree. Second, we present a method to analyse the fault trees using the SCT tool *Supremica* [5]. Finally, we present a method to calculate minimal cut sets using our approach. An advantage of our work is the compositional approach to modelling and verification that is beneficial in reasoning about large fault trees for highly complex

systems. To the best of our knowledge, SCT has not previously been used in the context of fault tree analysis.

The paper begins with a brief introduction to FTA and SCT in Sect. 2 and Sect. 3, respectively. Section 4 discusses modelling and analysis in *Supremica* with an example from the *Fault Tree Handbook* [16]. The paper is concluded with a brief discussion on future extensions in Sect. 5. Our work is successfully integrated with a model-based systems engineering tool [14], that is widely used in the automotive industry.

2 Fault Tree Analysis

Fault Tree Analysis (FTA) [16] is a top-down deductive safety analysis technique, where an undesired safety-critical failure of a system is specified, and then analysed in the context of its operational environment to find all possible ways in which the specified failure can occur.

A fault tree is a graphical model of various combinations of faults that cause the safety critical failure, represented as a top level failure event at the root of the fault tree. From this root event, the fault tree is constructed from a predefined set of symbols [16], which results in a set of combinations of component failures that can cause the top level failure. Note that the fault tree is not a model of all possible causes for system failure, but given a particular failure it depicts the possible combinations of basic component failures that lead to this failure. Since FTA is primarily a manual process, the exhaustiveness of the analysis is left to the assessment of the analyst.

Although several extensions of fault trees have been proposed [10], in this paper we limit ourselves to the symbols described in the *Fault Tree Handbook* [16]. Broadly, the nodes in the fault tree can be classified into three types: *events*, *gates*, and *transfer symbols* [16].

2.1 Pressure Tank System

The pressure tank system [16] in Fig. 1 describes a control system to regulate a pump-motor that pumps fluid into the tank. Initially the system is considered to be dormant and de-energized: switch S1 open, relays K1 and K2 open, and the timer relay closed. The tank is assumed to be empty in this state and therefore the pressure switch S is closed. It is also assumed that it takes 60 s to pressurize the tank, and an outlet valve, which is not a pressure relief valve, is used to drain the tank.

System operation is started by pressing switch S1. This closes and latches relay K1, and subsequently relay K2 to start the pump. When threshold pressure is reached, the pressure switch opens, causing K2 to open, and consequently the pump motor to cease operation. The timer allows emergency shut-down in case the pressure switch fails. Initially, the timer relay is closed and power is applied to the timer as soon as K1 closes. If the clock in the timer registers 60 s of continuous power, the timer relay opens and latches, thereby causing a system

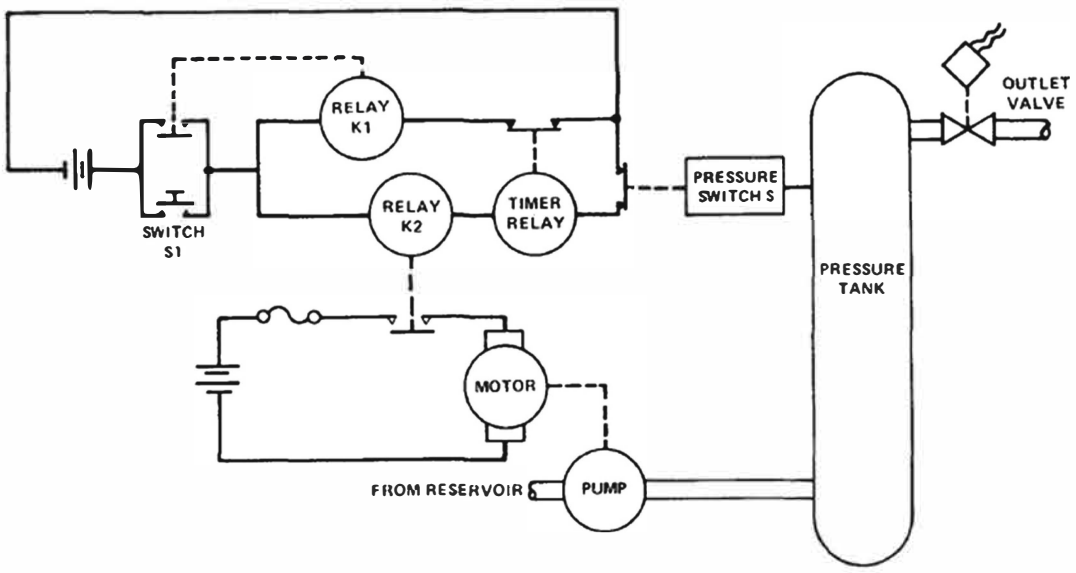


Fig. 1. Pressure tank system from [16], page VIII-1

shut-down. In normal operation, when pressure switch S opens, the timer resets to 0 s. When the tank is empty, the pressure switch closes, and the cycle can be repeated.

Figure 2 shows the basic fault tree from [16] (page VIII-13) for the pressure tank system. Here, the hazard 'rupture of pressure tank after start of pumping' is analysed and is represented by the top level failure event, E1. The basic events denoted by circles represent the respective component failures and form the leaves of the tree. The intermediate events, which are fault events that occur due to one or more antecedent causes are denoted by rectangles. The process of obtaining the fault tree following a top down analysis is out of scope of this paper; we assume a FT is given. A complete description of the example and the fault tree can be found in [16].

3 Supervisory Control Theory

The Supervisory Control Theory (SCT) [9] provides a framework to model, synthesize and verify control functions for *discrete event systems* (DES), which are dynamic systems characterised by the evolution of events causing the system to transit from one discrete state to another. Given a model of the system to control, a *plant*, and a *specification* describing the desired controlled behaviour, the SCT provides methods to synthesise a *supervisor* that dynamically interacts with the plant in a closed-loop, and restricts the event generation of the plant such that the specification is satisfied. The supervisor thus ensures a safe control of the plant by restricting the execution of certain events. However, only events that are *controllable* can be restricted by the supervisor, while events that are *uncontrollable* cannot be restricted. A dual problem that is of interest here, is

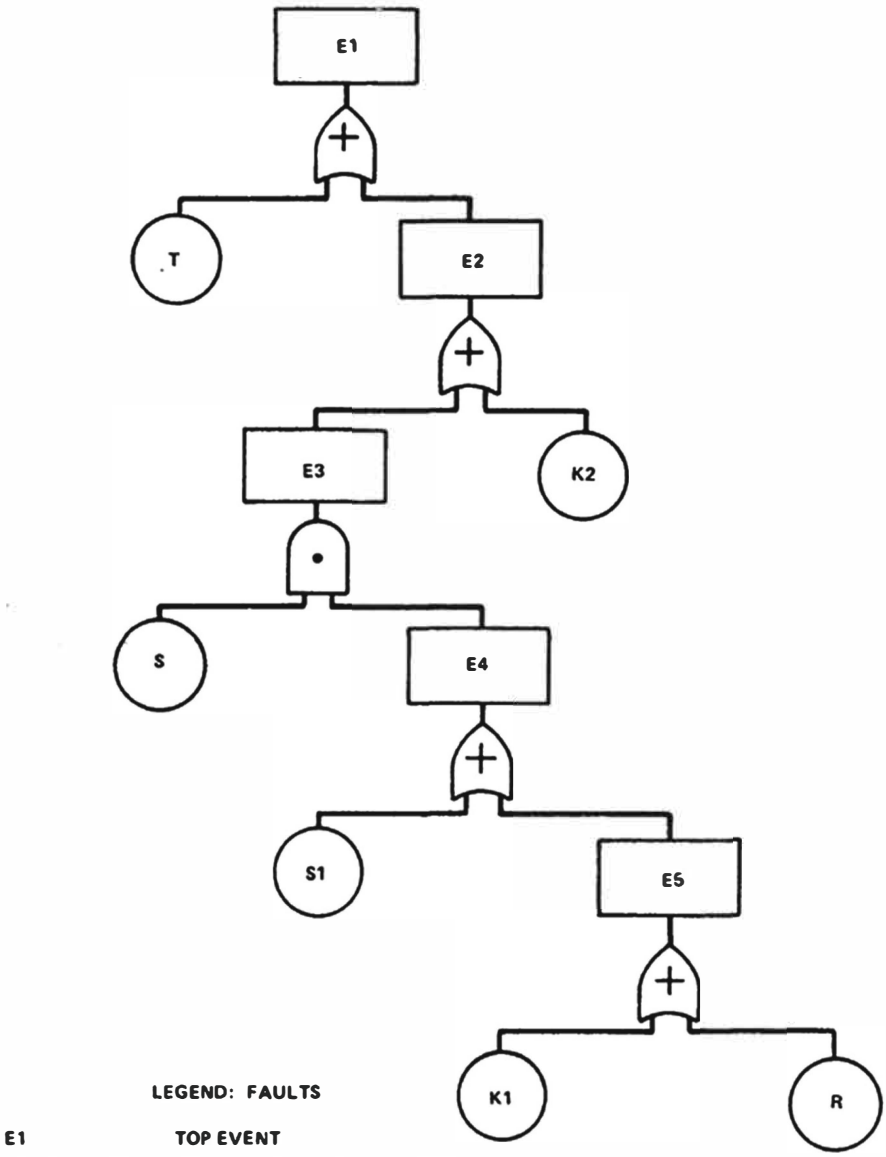


Fig. 2. Fault tree for pressure tank system in Fig. 1 from [16], page VIII-13

to given a model of a (controlled) plant and a specification, *verify* whether the specification is fulfilled or not. So, in this paper we use ideas from SCT to formally verify properties of the plant model, and do not focus on the synthesis of supervisors.

To model a fault tree as a DES, we use Extended Finite State Machines (EFSM) [13], which are finite state machines extended with bounded discrete *variables*, *guards* that are logical expressions over variables, and *actions* that assign values to variables on transitions.

Definition 1. *An Extended Finite State Machine (EFSM) is a tuple* $E = \langle \Sigma, V, L, \rightarrow, l^i, L^m \rangle$*, where* Σ *is a finite set of events,* V *is a finite set of bounded discrete variables,* L *is a finite set of locations,* $\rightarrow \subseteq L \times \Sigma \times G \times A \times L$ *is the conditional transition relation, where* G *and* A *are the respective sets of guards and actions,* $l^i \in L$ *is the initial location, and* $L^m \subseteq L$ *is the set of marked locations.*

A state in an EFSM is given by its current location together with the current values of the variables. The expression $l_0 \xrightarrow{\sigma:[g]a} l_1$ denotes a transition from location l_0 to l_1 labelled by event $\sigma \in \Sigma$, with guard $g \in G$, and action $a \in A$. The transition is enabled when g evaluates to *true*, and on its occurrence, the current location of the EFSM changes from l_0 to l_1, while a updates some of the values of the variables $v \in V$. EFSMs interact through shared events by *synchronous composition*, denoted $\mathcal{A}_1 \| \mathcal{A}_2$ for two interacting EFSM models, $\mathcal{A}_1$ and $\mathcal{A}_2$. In synchronous composition, shared events occur simultaneously in all interacting EFSMs, or not at all, while non-shared events occur independently. Transitions on shared events with mutually exclusive guards, or conflicting actions will never occur [13]. In an EFSM, *active events* are the events that label some transition, while *blocked events* do not label any transition. In the synchronous composition of two EFSMs, the blocked events of the synchronised EFSM, is the union of the blocked events of the synchronised EFSMs. That is, transitions in one EFSM labelled by events blocked by the other EFSM, will be removed.

3.1 Nonblocking Verification

Given a set of EFSMs $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, the *nonblocking* property guarantees that some marked state can always be reached from any reachable state in the synchronous composition over all the components $\mathcal{A}_i$. While the monolithic approach to nonblocking verification is explicit, it is limited by the combinatorial state-space explosion. The *abstraction-based compositional verification* [7] has shown remarkable efficiency to handle systems of industrial complexity. This approach employs *conflict-preserving abstractions* to iteratively remove redundancy and keeps the abstracted system size manageable. *Supremica* [6], a tool for modelling and analysis of DES models, implements the abstraction-based compositional algorithms (and others) for verification of EFSMs.

4 FTA in Supremica

In this section, we describe how the fault tree in Fig. 2 is modelled into a number of *plant* EFSMs. We demonstrate how the model can be validated by verifying

typical *specifications* in *Supremica*. This section also includes a brief discussion about computing minimal cut sets using our approach. Both *Supremica* and the models of this section are available online[1].

4.1 Modelling

To make the best use of compositionality, we incrementally model different failure events in a modular way. Given a fault tree, we first model the lowest level and gradually proceed towards the top level event. For the higher levels, we only consider the intermediate fault events from the lower levels and hide all other inner details.

Consider the lowest level of the fault tree in Fig. 2. It consists of two basic events as inputs to the lowest OR gate leading to the intermediate fault event, E5. This forms the first level in our modelling hierarchy. Fault event E5 can occur either due to a primary failure of K1 or a primary failure of R. This behaviour is modelled in the EFSM as shown in Fig. 3a. The two events $K1$ and R denote the corresponding primary failures and when either occurs, the EFSM transits from its initial location, A_0^i to location E_5[2].

With E5 modelled, we proceed to the next level, the intermediate fault event E4. From Fig. 2, we see that this can occur either due to a primary failure of switch S1 or due to the occurrence of E5. This gives us a total of 7 possible combinations that lead to E4. However, since we have modelled the analysis for E5 as an EFSM on the previous level, we can use guards to capture this, and model E4 with just 2 events as shown in Fig. 3b. The guard condition on the event $E5$ ensures that the event is enabled only in a situation where the EFSM in Fig. 3a is in location E_5. Here, the guard $[A_0 == E_5]$ represents that the current location of the EFSM A_0 in Fig. 3a, is E_5.

(a) EFSM modelling E5 (b) EFSM modelling E4

Fig. 3. EFSMs for intermediate failure events, E4 and E5 of the fault tree

The next level in our modular hierarchy is the output event of the only AND gate in the fault tree, E3. The two inputs to the AND gate correspond to the primary failure of the pressure switch S and the analysis resulting from the intermediate fault E4. Figure 4 shows the model for this fault event E3. Since the order of events do not matter in an AND gate, there are two possible ways to reach the failure state as shown in Fig. 4.

[1] https://supremica.org https://github.com/yuvrajselvam/FTA_SCT.

[2] In this paper, for a fault Ex in the FT, Ex denotes the corresponding event in the EFSM and E_x denotes the location reached due to the occurrence of the fault.

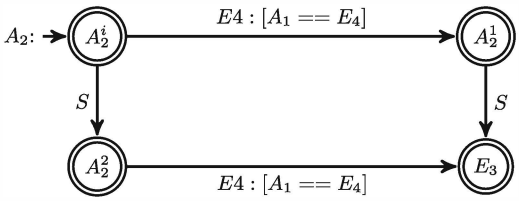


Fig. 4. EFSM, A_2 for the intermediate failure event E3

The final two levels of the fault tree corresponding to fault events E2 and E1 consist of OR gates and are modelled as already shown, see Fig. 5. Note that in the plant models, the only unmarked location is the initial location in Fig. 5b, and therefore in the synchronised plant model, which gives the complete fault tree, the marked locations correspond to the top level failure event E1.

(a) EFSM modelling E2 (b) EFSM modelling E1

Fig. 5. EFSM for intermediate failure events, E2 and E1 of the fault tree

For special cases of AND gates, like INHIBIT and PRIORITY-AND, the models look slightly different. For an INHIBIT gate, where the output is determined by a single input together with some qualifying condition, we can use a single event label together with the qualifying condition as a guard to model the transition to the failure state. For a PRIORITY-AND gate, where the output occurs only if all inputs occur in a specified ordered sequence, we can model the specified sequence as a path from the initial state to the failure state. For example if failure event E3 is at the output of a PRIORITY-AND with the order specified as E4 before S, then we only have the path $A_2^i \rightarrow A_2^1 \rightarrow E_3$ in Fig. 4 as the corresponding EFSM. This makes it possible to use EFSMs to model sequential dependencies as required by the PRIORITY-AND gate.

The distinction between inclusive and exclusive-OR gates can be ignored in the fault tree analysis when dealing with independent, low probability component failures (see [16], page VII-7). Therefore we do not introduce special approaches to differentiate them in our method. If a distinction is truly needed, additional guards and transitions can be introduced on the model.

Algorithm 1 presents a systematic method to construct EFSMs in a modular way from a given fault tree. Note that the algorithm includes modelling of two types of gates only, AND and OR. However, it can be extended to include other types of gates like INHIBIT and PRIORITY-AND as discussed above.

In Algorithm 1, lines 9–18 describe the modelling of OR gates and lines 19–30 describe AND gates. The addition of guards on the transitions mentioned in lines 16 and 28 describe the use of EFSM variables in guard conditions as shown in Fig. 3b for the OR gate, and in Fig. 4 for the AND gate, respectively.

4.2 Verification

In software controlled complex systems, safety analysis plays a significant role in formulating the safety requirements for the subsequent system design. Establishing confidence in the fault tree analysis is typically done manually. This is a shortcoming as it is error prone and even intractable for large and complex systems. An automated analysis method is very beneficial in providing sufficient verification evidence for the safety analysis phase. In this section, we present how typical specifications are modelled and verified using nonblocking verification algorithms in *Supremica*.

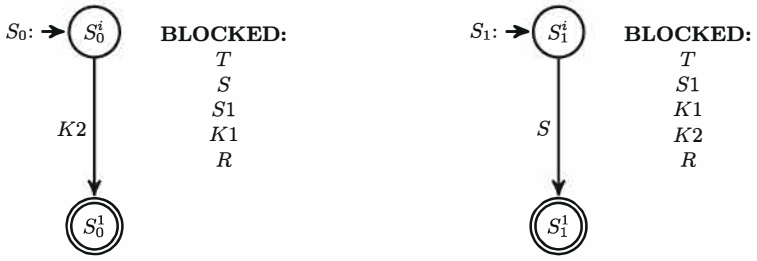
When system operation is started in the pressure tank in Fig. 1, the pump starts filling fluid into the tank. When the tank is full and the threshold pressure is reached, pressure switch S opens, causing K2 to open, and consequently the pump to stop. K2 failing to open would result in continuous pumping beyond the threshold and may result in the rupture of the tank. Therefore K2 is critical for safe operation and a primary failure of K2 may result in the top level failure event E1. Ideally, this behaviour should be captured in our FTA and we can verify this. Figure 6a shows the EFSM modelling this specification. $K2$ is the only active event in this EFSM and the other basic events in the fault tree are blocked. Recall that transitions labelled by blocked events are removed in the synchronous composition of the specification and the plant models. Therefore, by blocking all basic events but $K2$, we ensure that $K2$ is included in the marked language of the EFSM whereas other basic events are not. A nonblocking verification performed on the synchronised model of this specification together with the plant models, shows that the system is nonblocking, thereby verifying that a primary failure of K2 is sufficient to cause rupture of the tank, the failure event E1.

On the other hand, since we have the timer relay as a backup in the system, only a failure of the pressure switch, S, should not lead to tank rupture. We can model this as a specification shown in Fig. 6b. Since we are only interested in the primary failure of pressure switch S, we block the remaining basic events in the fault tree. A nonblocking verification of this specification synchronised with the plant model results in a blocking state, thereby verifying that only S occurring will not result in the top level failure event E1. However, if we also include the failure of the timer relay R, we get the specification as shown in Fig. 7. With this specification, we can verify that the system is indeed nonblocking, i.e., a failure of both components S and R will lead to the top level failure event E1. Specifications to model the remaining causes leading to the top level event and/or the intermediate events are done in a similar way as in Figs. 6 and 7.

Algorithm 1: Modular fault tree modelling

Input: Fault Tree, FT
Output: EFSM set corresponding to the fault tree, FT

```
Initialisation                                                                  1
    declare basic events set, BE                                                2
    declare variables, Q, curr_node, child                                      3
add root (FT) to Q // queue, Q contains elements to be processed                4
BE:= getBasicEvents (FT)                                                        5
while Q ≠ ∅ do                                                                  6
    curr_node:= pop (Q) // get the oldest element in queue                      7
    gate:= getGate (curr_node) // retrieve connecting gate of node              8
    if gate is OR then                                                          9
        create initial and terminal locations, l_0 and l_n                      10
        foreach child ∈ getChildren (gate) do                                   11
            if child ∈ BE then // child is a basic event                        12
                addTransition(l_0, l_n, child)                                  13
            else // child is an intermediate event                              14
                addTransition(l_0, l_n, child)                                  15
                add guards using automaton variables on the respective transitions 16
                add child to Q                                                  17
        markLocations(curr_node, root (FT))                                     18
    else // node is an AND gate                                                 19
        create initial and terminal locations, l_0 and l_n                      20
        children:= getChildren (gate)                                           21
        create a set of strings, 𝕊, by permutation over children                22
        // each string is a path from l_0 to l_n
        foreach string ∈ 𝕊 do                                                   23
            create transitions and locations correspondingly                    24
            obtain the set of events, 𝔼                                         25
            foreach event ∈ 𝔼 do                                                26
                if event ∉ BE then // it is intermediate event                  27
                    add guards using automaton variables on respective transitions 28
                    add event to Q                                              29
        markLocations(curr_node, root (FT))                                     30
function markLocations(curr_node, root (FT))                                    31
    if curr_node == root (FT) then                                              32
        mark the terminal location, l_n                                         33
    else                                                                        34
        mark all locations                                                      35
function addTransition(l_a,l_b,event)                                           36
    add transition between l_a and l_b                                          37
    label transition with event                                                 38
```



(a) EFSM for $K2 \rightarrow E1$

(b) EFSM for $S \rightarrow E1$

Fig. 6. EFSM for specifications

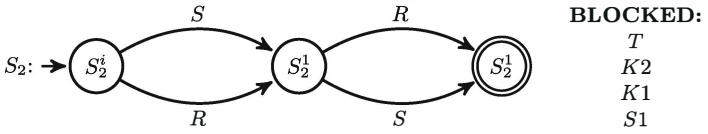


Fig. 7. EFSM for specification $S \wedge R \rightarrow E1$

The type of specifications that we have seen so far are modelled to check whether certain basic events or combinations of events lead to a failure event. Given such a specification, SP, and fault tree, FT, Algorithm 2 presents how EFSM models can be obtained from them.

4.3 Minimal Cut Sets

Our approach is not only useful for verification but also in calculating minimal cut sets, one of the most prominent qualitative analysis techniques of standard fault trees. A *cut set* is a set of component failure events that together lead to the top level failure. Formally, a minimal cut set is a *smallest* combination of component failures which, if they all occur, lead to the top level failure event. It is smallest in the sense that all failures are needed for the top level event to occur and if one of them in a cut set does not occur, then the top event will not occur by that set. For example, the minimal cut sets for the pressure tank system are {T}, {K2}, {S, S1}, {S, K1}, {S, R}.

In our modelling approach presented in Sect. 4.1, the marked locations in the composed model correspond to the top level failure event. This makes it possible to use the marked language of the plant EFSM to calculate the minimal cut sets. In our case, a cut set is a set of events that lead to marked locations corresponding to the top level failure event. Calculating minimal cut sets is then done by finding the shortest paths in the synchronised plant EFSM from the initial location to the marked locations, a task typically solved by variants of breadth-first search algorithms. Algorithm 3 presents one such method to calculate minimal cut sets by exploiting the marked language of the synchronised EFSM. Lines 11–13 of

Algorithm 2: Modelling specifications

Input: Fault Tree, FT and Specification, SP
Output: EFSM modelling the specification
Initialisation 1
  declare basic events set, BE 2
  declare active events set, AE 3
  declare blocked events set, BLOCKED 4
BE:= `getBasicEvents` (FT) 5
AE:= `getBasicEvents` (SP) 6
create locations $l_0, l_1, ..., l_N$ with $N = |\text{AE}|$ 7
make l_0 the initial location 8
make l_N the single marked location 9
for every pair (l_{i-1}, l_i) with $i \in \{1, 2, ..., N\}$ create N transitions 10
label each transition uniquely from $\sigma \in$ AE 11
add blocked events, BLOCKED:= BE $\setminus$ AE 12

the algorithm adds the basic events that can reach the marked location in the synchronised EFSM to the output set. Lines 14 and 15 ensure that the same events are not repeated.

Algorithm 3: Computation of Minimal Cut Sets

Input: $\text{EFSM}_1, \ldots, \text{EFSM}_n$ modelling the considered FT
Output: Set of minimal cut sets, S
Initialisation 1
  declare variable Q as queue with states to be processed 2
  declare synchronised EFSM A as $\text{EFSM}_1 \,||\, \ldots \,||\, \text{EFSM}_n$ 3
  declare basic event set, BE 4
  declare blocked events set, BLOCKED 5
BE := `getBasicEvents`(A) 6
while $\exists e \in \{\sigma \mid \exists s' \ s.t. \ (s_i, \sigma, s') \in \rightarrow_A \wedge \sigma \in \text{BE}\}$ **do** 7
  Q.`put`(s_i) `// Enqueue the initial state` s_i 8
  while Q $\neq \emptyset$ **do** 9
    s := Q.`get`() `// Dequeue state` s `from Q` 10
    if $\exists s', \exists \sigma \ s.t. \ (s, \sigma, s') \in \rightarrow_A \wedge$ `isMarked`(s') **then** 11
      `// Retrieve basic events labelling transitions from` s_i `to` s
      Σ_c := `getEvents`(s_i, s') $\cap$ BE 12
      `//` Σ_c `is one minimal cut set, insert it into S`
      S.`put`(Σ_c) 13
      create a single location (marked) EFSM_{sp} with BLOCKED := Σ_c 14
      `// Update A by blocking all basic events in` Σ_c
      A := A $||$ EFSM_{sp} 15
      break 16
    else 17
      forall the $s' \ s.t. \ (s, \sigma, s') \in \rightarrow_A$ **do** Q.`put`(s') 18

5 Conclusion

We have shown how fault tree analysis can be formalised to be automatically analysed by modelling techniques from Supervisory Control Theory (SCT) using the tool *Supremica*. We present a systematic approach to incrementally obtain formal models from a given standard fault tree, as summarised in Algorithm 1. Algorithm 2 describes a method to automatically generate specifications for given properties of the fault tree, so that these properties can be verified using non-blocking verification. Finally, Algorithm 3 presented a method to automatically calculate minimal cut sets from the generated models.

Though our modelling approach can model complex systems with redundant architectures and dynamic dependencies, we here limit ourselves to the standard symbols described in the *Fault Tree Handbook*. Our approach can indeed be extended to use *dynamic* gates. The formal model obtained from the approach discussed in this paper, considers only the fault behaviour of the system as described by a given fault tree and nothing else. While we verify certain properties on the model to establish confidence in the system, we do not focus on correctness of the construction of the fault tree in the context of the system's operational environment. In a behavioural approach, we would formally model the complete behaviour of the system, i.e., including the nominal operational behaviour and not only the fault behaviour. This presents a wide range of possibilities. One possible extension is to adopt a formal approach similar to model checking [15]. Another notable extension of our work is to use the behavioural system models and the supervisor synthesis framework provided by SCT to automatically synthesize the fault behaviour. This falls in line with the model-based dependability analysis [12] approach for safety analysis. In such extensions, the system model becomes the plant models and the work in this paper can then be used to obtain formal specifications from a given fault tree. This approach makes it possible to use such formal models in several stages of a model-based design process. The state based models that are created can be re-used during the development of the software programs in the later stages. The work presented in this paper can provide a solid basis for possible extensions in those areas.

A primary motivation for this work is our current focus on formal verification of autonomous driving systems where SCT and *Supremica* have been used to verify software for autonomous driving systems [11]. We believe our work in this paper will strongly encourage the application of SCT and *Supremica* in different stages of safety critical software development starting from safety analysis in the early stages to synthesis and verification of the software in the end stages. Our work in this paper is successfully integrated with a model-based systems engineering tool [14], that is widely used in the automotive industry.

References

1. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans. Reliab. **41**(3), 363–377 (1992)

2. Hansen, K.M., Ravn, A.P., Stavridou, V.: From safety analysis to software requirements. IEEE Trans. Softw. Eng. **24**(7), 573–584 (1998)
3. Kabir, S.: An overview of fault tree analysis and its application in model based dependability analysis. Expert Syst. Appl. **77**, 114–135 (2017)
4. Kaiser, B., Gramlich, C., Förster, M.: State/event fault trees–a safety analysis model for software-controlled systems. Reliab. Eng. Syst. Saf. **92**(11), 1521–1537 (2007)
5. Malik, R.: Programming a fast explicit conflict checker. In: 2016 13th International Workshop on Discrete Event Systems (WODES), pp. 438–443. IEEE (2016)
6. Malik, R., Akesson, K., Flordal, H., Fabian, M.: Supremica-an efficient tool for large-scale discrete event systems. IFAC-PapersOnLine **50**(1), 5794–5799 (2017). https://doi.org/10.1016/j.ifacol.2017.08.427
7. Mohajerani, S., Malik, R., Fabian, M.: A framework for compositional nonblocking verification of extended finite-state machines. Discrete Event Dyn. Syst. **26**(1), 33–84 (2015). https://doi.org/10.1007/s10626-015-0217-y
8. Palshikar, G.K.: Temporal fault trees. Inf. Softw. Technol. **44**(3), 137–150 (2002)
9. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
10. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15**, 29–62 (2015)
11. Selvaraj, Y., Ahrendt, W., Fabian, M.: Verification of decision making software in an autonomous vehicle: an industrial case study. In: Larsen, K.G., Willemse, T. (eds.) Formal Methods for Industrial Critical Systems, pp. 143–159. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-27008-7_9
12. Sharvia, S., Kabir, S., Walker, M., Papadopoulos, Y.: Model-based dependability analysis: state-of-the-art, challenges, and future outlook. In: Software Quality Assurance, pp. 251–278. Elsevier (2016)
13. Skoldstam, M., Akesson, K., Fabian, M.: Modeling of discrete event systems using finite automata with variables. In: 2007 46th IEEE Conference on Decision and Control, pp. 3387–3392. IEEE (2007)
14. SYSTEMITE: Systemweaver. https://www.systemweaver.se/. Accessed 09 May 2020
15. Thums, A., Schellhorn, G.: Model checking FTA. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 739–757. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_40
16. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault tree handbook. Technical report, Nuclear Regulatory Commission Washington DC (1981)
17. Xiang, J., Ogata, K., Futatsugi, K.: Formal fault tree analysis of state transition systems. In: Fifth International Conference on Quality Software (QSIC 2005), pp. 124–131. IEEE (2005)