State-Event Fault Trees - A Safety Analysis Model for Software Controlled Systems

Bernhard Kaiser, Catharina Gramlich, Marc Förster

Fraunhofer Institute for Experimental Software Engineering,
Sauерwiesen 6, Kaiserslautern, Germany
{bernhard.kaiser, marc.foerster}@iese.fraunhofer.de

Hasso Plattner Institute for Software Systems Engineering,
Department of Software Engineering and Quality Management,
Prof.-Dr.-Helmert-Str. 2-3, Potsdam, Germany
catharina.gramlich@hpi.uni-potsdam.de

Abstract. Safety models for software-controlled systems should be intuitive, compositional and have the expressive power to model both software and hardware behaviour. Moreover, they should provide quantitative results for failure or hazard probabilities. Fault trees are an accepted and intuitive model for safety analysis, but they are incapable of expressing state dependencies or temporal order of events. We propose to combine fault trees with an explicit State/Event semantics using a graphical notation that is similar to Statecharts. Our new model, named State-Event Fault Trees (SEFTs), subsumes both deterministic state machines suited to describe software behaviour, and Markov chains that model probabilistic failures, while keeping the visualisation of causal chains known from fault trees. We allow exponentially distributed probabilistic events, deterministic delays and triggered events. The model is compositional and joins components by typed ports. Quantitative evaluation is achieved by translating the component models to Deterministic and Stochastic Petri Nets (DSPNs) and using an existing tool for analysis. This paper, which is an extended version of [17], revisits the model elements and the analysis procedure and provides a small case study of a fire alarm system, completed by an outlook on our tool project ESSaRel.

1 Introduction

In technical systems, more and more mechanical and electrical components are replaced with software-controlled components. This includes safety critical domains such as avionics, automotive or industrial control. In these application fields safety and reliability analysis is a mandatory part of the development and must be supported by appropriate models and tools. Fault Tree Analysis (FTA) is one of the most widely used techniques in this context. Fault Trees (FTs) are intuitive for practitioners due to their hierarchical structure and the familiar logical symbols. They provide a set of qualitative and quantitative analyses. FTs have been used for several decades in the context of mechanical or electrical systems and are gaining importance in the context of software-controlled systems.
Nevertheless, some fundamental differences between fault trees and the models commonly used for embedded system design are obvious: First, models for complex systems must be compositional. Modularisation of FTs, however, is only defined in a restricted way, based on independent subtrees. Second, safety is principally a matter of behaviour and, in contrast to the state-space models used in systems design, FTs are not suitable for modelling behaviour. FTs are a combinatorial model that cannot capture sequences of actions and state history. Moreover, the two-state abstraction (either working or failed) of fault trees is not adequate for systems with complex state spaces.

These differences not only hamper the application of FTA to software-controlled systems, but also obstruct the integration of state-based submodels into an FTA. This integration is desirable for two reasons: first, because it would enable reuse of state-based models from the design phase for the explanation of component behaviour during safety analysis, and second, because it would allow the integration of Markov chains, which are an important state-based safety and reliability model.

Existing approaches to overcome the semantic weaknesses of FTs often rely on formal methods that are not familiar to practitioners and do not offer visual integration for FTs and state-based models.

We take a different approach by adding a notion of states and events to FTA: States describe conditions that last over a period of time whereas events are sudden phenomena, including state transitions. We call this extended model State-Event Fault Trees (SEFTs). States and events are depicted by different symbols. We propose typed FT gates for states and events (e.g. an OR gate with two event inputs and another OR gate with two states inputs). Regarding the AND gate that joins two events we distinguish a History AND that remembers events that have occurred in the past and a Priority AND that additionally remembers if they have occurred in a given order. SEFTs are partitioned into components which are interconnected by ports. This port concept allows the integration of component models other than fault trees, such as Markov chains or state diagrams from CASE tools.

SEFTs are well suited for industrial use since they unite familiar graphical notations borrowed from FTA and Statecharts; nevertheless, their semantics allows quantitative analysis. Analysis is performed by component-wise translation of the SEFT models into Deterministic and Stochastic Petri Nets (DSPNs) [6] a class of Petri nets for which analysis tools exist – e.g. the TimeNET tool [23]. In the Petri net domain the component models are merged into one flat model that is passed to an existing analysis tool.

In this paper we explain the application of SEFTs and the steps necessary for their translation to DSPNs. To illustrate the procedure we refer to a small case study of a fire alarm system. The rest of the paper is organised as follows: In Section 2 we give a short overview over FTA and previous adaptations to software-controlled systems. In Section 3 we introduce the modelling elements of SEFTs in summary and explain the analysis by translation to DSPNs. In Section 4 we introduce the case study and show how SEFT analysis is applied in practice. Section 5 describes the research tool ES-SaRel (Embedded Systems Safety and Reliability Analyser) [7] which implements the algorithm for SEFT analysis and is a successor of the FTA tool UWG3. Section 6 concludes the paper and gives some pointers to ongoing and future research steps.
This paper is an extended and revised version of [17] and incorporates recent preci-
sions and amendments published in [15] [16].

2 Foundations and Previous Work

2.1 Introduction to Fault Tree Analysis

FTs [22] are a widely accepted model that graphically shows how influence factors
(in general component failures) contribute to some given hazard or accident. They
provide logical connectives (called gates) that allow decomposing the system-level
hazard recursively. The most fundamental gates are the AND gate and the OR gate.
The AND gate indicates that all influence factors must apply together to cause the
hazard and the OR gate indicates that any of the influences causes the hazard alone.
Both gates are shown in Fig. 2-1, in the IEC 61025 notation and in the more common
US notation. In this paper, as well as the ESSaRel tool, we use a gate notation simi-
lar to standard IEC 61025, since these symbols can be adapted more easily to new
gate types.

Fig. 2-1: AND and OR gate in IEC 61025 (left) and US (right) notation

The logical structure is usually depicted as an upside-down tree with the hazard to
be examined (called top event) at its root and the lowest-level influence factors (called
basic events) as the leaves (see Fig. 2-2). Note that in the context of FTA the term
“event” is applied in its probability theory meaning: an event is not necessarily some
sudden phenomenon, but can be any proposition that is true with a certain probability.

Fig. 2-2: Traditional fault tree
Analyses to be performed on FTs can be qualitative or quantitative. Qualitative analyses list, for instance, all combinations of failures that must occur together to cause the top-level failure. Quantitative analysis calculates the probability of the top-event from the given probabilities of the basic events. Combinatorial formulas indicate for each type of gate how to calculate the output probability from the given input probabilities. These probabilities are either probabilities that an event occurs at all over a given mission time or they are understood with respect to a given point in time. The evolution of a system over time or any dependencies between the present system behaviour and the history cannot be modelled. An important assumption in order to obtain correct results is the stochastic independence of the basic events, which is hard to achieve in complex networked systems. Most current FTA tools use the efficient representation of Boolean terms as Binary Decision Diagrams (BDDs) to compute the quantitative results.

2.2 Recent Research on Fault Tree Analysis for Software-Controlled Systems

Like many safety and reliability analysis models, FTs were originally designed for non-programmable systems. When more and more technical systems became software-controlled, the need to adapt FTs to this application field grew. There have been several attempts to adapt FTA to software or embedded systems, to derive FTs from software models and to enhance the expressive power of FTs. [20] integrate FTs with formal program specifications and use Interval Temporal Logic to give a formal semantics to Fault Trees. Formal methods are also used in [2] and [10]. Other approaches to modelling dynamic behaviour and multi-state components map FTs to Markov chains [2] or different variants of Petri nets [4] [5] [11] [14]. Some researchers [10] [2] proposed additional fault tree gates, describing for instance conditional probability, sequence enforcing or various spare usage situations (hot, cold and warm spare) in order to model special cases of dependencies.

For an efficient and sound development process, different modelling techniques from system design and safety/reliability analysis should smoothly integrate with each other. Research projects aiming at the integration of different models can increasingly be observed during the last years [8] [3]. Many of them consider FTs, but often they are applied in a rather informal or qualitative way.

2.3 State-Based Modelling Techniques in Software and Reliability Analysis

The most important family of models used to describe embedded systems behaviour are the state-based models. There is a large set of deterministic and probabilistic models that share the concept of a finite set of states and a finite set of events that mark the transitions between states. In the systems design phase their purpose is to describe deterministic behaviour – the desired behaviour of the system. The state machine variants used for this purpose usually annotate the transitions with the name of the event or the system input that triggers (causes) the transition and optionally a condition (“guard”) that must be true in order to allow the state transition. Additionally, an output action that is performed on the state transition can be annotated. The triggers determine when and – in conjunction with the guards – to which target state a transition takes place. A small example can be seen in Fig. 2-3. There are two states, A and B. If the component is in state A and trigger event X occurs, and at the same time
condition c is true, then the component changes its state to B. If trigger event Y occurs in state B, then the component goes back to state A.

![Basic state diagram](image)

**Fig. 2-3: Basic state diagram**

One of the most popular state-machine models applied in software and systems engineering is Statecharts. Statecharts have been introduced by [13] and have been adopted and modified in many modelling techniques. Statecharts annotate triggers, guards and actions to state transitions. Additionally, they introduce a concept of hierarchy by decomposing states into substates and provide a broadcast communication concept, e.g. each event in the system can be observed immediately in each part of the system.

ROOMcharts (ROOM – Realtime Object Oriented Modelling), introduced by [21], further extend the Statechart approach by introducing the notion of components that are joined by ports and communication by messages that are exchanged via these ports.

Probabilistic state-machines are an important modelling technique in reliability, performance and timing analysis. The most basic and most important probabilistic state-based model is the Markov Chain (MC). A Markov chain is a state machine whose transitions are labelled with transition rates, i.e. conditional probabilities that the transition to a given successor state occurs in the next small time interval, provided that the system is in the source state. A set of analytical and numerical solution techniques exists; simulation is also possible to obtain sojourn probabilities in certain states or average sojourn times. A disadvantage of MCs is that a commonly accepted modularisation concept does not exist and thus models quickly become unmanageable.

### 2.4 Component Fault Trees

Compositionality is an essential feature when dealing with highly complex systems. Compositional models recursively decompose a system into components, subcomponents and so on, leading to a hierarchy of components. A common paradigm in compositional models is to show the inner details of a component on its own hierarchical level, but to hide them on the next higher hierarchical level. On this higher level, all referenced subcomponents appear as “black boxes”. The relations, namely communication relations, between these black boxes are depicted as edges (lines). The points that connect these edges to a box are usually called ports. The ports of a component specify its external interface; only via these ports internal details of a component can be accessed from outside. In different models, ports are typed according to appropriate principles, e.g. by type and direction of the messages that can be exchanged across the port in a communication relation. What exactly is exchanged via ports depends on
the semantics of the actual model. Most current system design models provide this kind of compositinality.

![Fault tree with traditional structure and a subcomponent that is no module](image)

Fig. 2-4: Fault tree with traditional structure and a subcomponent that is no module

Traditional FTs, although they represent a hierarchical decomposition of failure causes, do not provide this kind of compositinality. They are compositional only in the sense that independent subtrees (called *modules*) can be cut off and handled separately. Technical components, however, are often influenced by other components and thus cannot be modelled by independent subtrees. An example is given in Fig. 2-4. Suppose the part within the dashed box corresponds to a technical subcomponent, possibly delivered by a sub-supplier and the rest is considered to be the main system. The subcomponent is dependent on events from the main system and therefore no module in the sense of FTA; it cannot be modelled by a subtree on its own.

![CFT of the System (left) and the subcomponent (right)](image)

Fig. 2-5 CFT of the System (left) and the subcomponent (right)

In order to achieve suitable modularisation in these cases, we recently proposed a more advanced component concept [18]. We call this enhanced model Component Fault Trees (CFTs). It allows cutting arbitrary parts out of a fault tree so they can be modelled and stored independently, leading to a modularisation that reflects the actual
technical components. Component models can be edited by different people and stored for later reuse in other projects. The model is integrated and flattened during analysis.

The CFT concept is introduced by taking the small example from above further: on main system level the subcomponent appears as a black box. The connection of the box to its environment is achieved by ports, points of information transfer. These ports appear as small symbols at the edge of the box. As in the compositional design models the inner details of the subcomponent are hidden on system level and put into a separate CFT model. In this model the ports reappear, this time as triangular transfer symbols (see Fig. 2-5). Of course, there can be more than one subcomponent on each level, and the hierarchy depth can be greater than two (i.e. a subcomponent can have sub-subcomponents and so on).

In this example, neither the system nor the subcomponent are analysable on their own, because they have open ports where information is missing. Nevertheless, it is possible to store the subcomponent independently, to deliver it together with other component models, to instantiate it several times and to keep it in a repository for later reuse. Additionally, some transformation and simplification steps can be performed on component level and pay off each time the component is used. We recently proposed a component-aware BDD (Binary Decision Diagram) algorithm that exploits the component concept to reduce BDD size [19]. The final quantitative result however can only be calculated if the components on all hierarchy levels are available.

From a semantical point of view CFTs are ordinary FTs with the mentioned restrictions. However, apart from the better compositionality, the CFT concept prepared the ground for the use of ports to achieve integration of other models. We later refined our ports into state ports and event ports, as will be explained in detail in the following. When we started to integrate components, described by Markov chains or Statecharts, as subcomponents into CFTs we found that the lack of semantic precision of FTs made it hard to connect states or events consistently to an FT. In response we took the approach of enhancing FTs by a state/event distinction, in order to allow the combination of different modelling elements and techniques, underpinning the model by a state-machine semantics. This led to the concept of State-Event Fault Trees.

3 State-Event Fault Trees

3.1 Introduction to the SEFT Notation Elements

State-Event Fault Trees (SEFTs) are a visual notation combining elements from FTA and from Statecharts [13], ROOM charts [21] or similar notations. The main difference to traditional FTs is the visual distinction of states and events. SEFTs are targeted for safety and reliability analysis and the focus is on the representation of causal chains, as in standard FTs. The underlying intention is to create an intuitive, but unambiguous and analysable modelling technique for industrial practitioners and to hide the formalism that lies behind. Mathematical notations that would be hard to understand for practitioners are avoided, where possible.
In contrast to Statecharts or ROOMcharts that depict transitions by directed arcs, an explicit event symbol is introduced. This is necessary in order to allow causal edges from events or states to other events to depict cause-effect relations (triggering and guarding). These causal edges correspond to the edges in traditional FTs and can be connected by logical gates, as usual in FTA.

The component concept developed for CFTs has been further developed for SEFTs: each system (or top-level component) may be decomposed into subcomponents. Components are self-contained and concurrently acting entities. Components are connected via ports that are typed as state ports or event ports in SEFT. This abstraction can generally be applied to both software and hardware; thanks to the probabilistic modelling elements even system users’ behaviour or probabilistic scenarios in the system environment can be modelled.

We deal with a finite state space for each component, an abstraction that is sufficient for safety and reliability considerations. We denote states by rounded rectangles, as in Statecharts. Each component is in exactly one state at each instant of time, called the active state (we leave out state hierarchy for now) and stays in that state for some time interval. So for any point in time $t$ the proposition “Component C is in state S at time $t$” is either true or false. Any such proposition is called a state expression or condition. State expressions may be atomic or composed by means of the standard Boolean connectives (AND, OR, NOT...). When performing the probabilistic analysis of an SEFT, a probability in the interval $[0, 1]$ for each given point of time can be assigned to a state expression.

![Diagram](https://via.placeholder.com/150)

**Fig. 3-1** a) Event with exponentially distributed delay  

b) Deterministically delayed event  

c) Triggered event

*Event* is the term we use for atomic phenomena that do not take time to occur (this is in contrast to the standard FT definition). In particular, state transitions are events, but there may be independent events as well, e.g. spontaneous actions that occur in the environment (e.g. “tube breaks”). For quantitative analysis a probability density must be assigned to events. If an event is a transition from one state to another we call these states predecessor and successor state. We distinguish the event (denoting a class of similar phenomena that can occur at different times) from the occurrence, which is associated with an instant of time. Since we refer to a continuous time scale for our model we assume that any two independent events cannot occur at exactly the same time.

We mark events by solid bars. The resemblance to Petri net transitions is not coincidental: we later translate events to Petri net transitions for analysis. In our model events occur in one of three ways: either they are triggered by other events, or they occur after a deterministic delay $t$ or an exponentially distributed probabilistic delay.
upon entry of their predecessor state. Thus, the expressive power of SEFTs subsumes both Statecharts that appropriately describe software behaviour and Markov chains that are a customary state-based model for hardware failures. Fig. 3-1 shows all of these variants.

The example of the triggered transition is simplified for explanation purpose; it ignores the fact that the states and events belong to different components. Note that there are two different kinds of directed edges: Those with light arrowheads mark the predecessor-successor relation between states and events (temporal edges) and those with bold arrowheads mark the triggering relation (causal edges). Causal edges between two events signify that each time the source event occurs, the target event occurs as well, provided that it is enabled. Enabled means that the component the target event belongs to is in one of the predecessor states of the target event. If the source event happens at an instant $t$, then the target event occurs at $t'$, so triggering does not encompass any delay. If, however, the modeller wants to introduce some explicit deterministic or probabilistic delay between source and target event, SEFTs offer a Delay gate. Causal edges can also have states as their source. States cannot trigger other states or events, but state terms can serve as guards for events, meaning that the event can only occur if the state term evaluates to true. Fig. 3-2 shows the basic SEFT symbols.

As in FTA, gates add logical connectors to the causal paths. Consequently the edges that connect gates are called causal edges as well. The most important gates are AND, OR and NOT in their different variants. SEFT gates are typed in the sense that they have different semantics depending on whether they are applied to state terms or to event triggering relations. For instance, the fragment in Fig. 3-3 has the semantics that the event “Pressure exceeds critical level” triggers the event “Boiler explodes” only if the state term “Safety Valve is defective” or the term “Pressure Sensor is defective” is true. In a complete example these unspecified state terms (drawn as dotted state symbols) could be states of two other components “Valve” and “Sensor”.

![Basic symbols of the SEFT component concept](image-url)
SEFTs allow the extension of fault trees to Directed Acyclic Graphs (the same cause triggering multiple effects) and deal with repeated events or states correctly. Causal loops (i.e. cycles in sequences of causal edges) are forbidden, except if some explicit delay is introduced into the cycle. This is not a severe restriction to expressiveness, because traditional FTs do not allow loops either and in practical cases always some deterministic or probabilistic delay is present. Forbidding causal loops keeps unnecessary semantical questions about the model of computation out of the SEFT model. The absence of causal loops must be checked before the analysis is started.

Just as Component Fault Trees, SEFTs are organized by components. Components are prototypes and must be instantiated. Components can be referenced as subcomponents of another component, forming a component hierarchy. The component on top of the hierarchy is the system to be examined. Each instance of a component defines a separate name space and all internal states and events are distinct from the state and events of other instances and hidden from the environment.

Ports connect components across hierarchy levels. We distinguish input ports from output ports and state ports from event ports. Subcomponents appear as black boxes where only the ports are visible. Examples can be found in the case study in section 4. Event ports allow triggering relations from one component to another. The information that some event occurs is transferred from the source component (the component where the output port belongs to) to the destination component (the component possessing the input port). There it can provoke some effect, provided that the destination component is ready to accept it. Otherwise, the event in the source component is neither blocked nor stored, but just discarded. The semantics of a state port is that the destination component has access to the information whether or not the state term in the source component is active, without having any means to influence that state.
3.2 Application of SEFTs

SEFTs are constructed like traditional FTs. Starting with some undesired system state (hazard) or event (accident), the analyst traces back its influences and finds out which system states or events play a role in initiating, propagating or inhibiting the fatal behaviour. The richer variety and semantic precision of gates allows to better capture chains of embedded systems behaviour. The basic events of standard FTA correspond to solitary1 exponentially distributed events in SEFTs. The project is structured hierarchically using the component concept.

Models that explain the relevant behaviour of subcomponents can be plugged in where necessary. For stochastic failures Markov chains are appropriate. They have traditionally been used for hardware wear-and-tear, but there can also be stochastic models for software failures. To model software and control aspects of the system, Statecharts or similar models from the design phase can be reused, e.g. by importing them from a CASE tool. The visible difference is that transitions, which are originally represented by labelled edges, now appear as explicit transition symbols. Software models can serve to check the reaction of the correct software on rare or unforeseen events from the environment, or probabilistically model software failures.

3.3 Analysis by Translation to DSPNs

A model is more useful if it not only provides a syntactically defined graphical notation but also a defined semantics. This makes it possible to provide analyses for relevant properties, supported by software tools. However, defining a formal semantics for a human-centred notation is a difficult task, as various attempts to formalise Statecharts or FTs show. A different issue is that, being state-based models, SEFTs cannot be evaluated by the traditional combinatorial FTA algorithms. To tackle both issues at the same time we propose to translate SEFTs into an accepted formal notation for which known analysis algorithms exist; this gives a definition for the semantics and provides a way towards quantitative analysis.

Petri Nets (PNs) are a model for discrete state systems that supports the concurrency we have to deal with in component based systems and provides stochastic variants. We chose Deterministic and Stochastic Petri Nets (DSPNs) [1] since they possess all kinds of transitions we need and provide analysis techniques for the properties we are interested in (in particular the probability of a place to be marked). They are an extension of Generalized Stochastic Petri Nets (GSPNs) that lack deterministic delay that often has to be considered in software behaviour. Assuming some basic knowledge about Petri nets we briefly point out the main features of DSPNs:

DSPNs are a timed variant of Petri nets, i.e. the (possibly probabilistic) time a transition waits before it fires after becoming enabled is specified in the model. In particular, DSPN transitions fire in one of three ways: immediately on activation, after a constant delay (specified by an annotated time parameter) or after an exponentially distributed probabilistic delay (specified by an annotated rate parameter). Firing of transitions is atomic and takes no time. In the graphical representation, black bars

1 A solitary event is an event without predecessor and successor state. An exponential solitary event is a shorthand for a two-state Markov chain and an event with the given rate from initial state to failed state. Thus it corresponds to an exponential basic event in standard FTA.
depict immediate transitions, empty rectangles depict transitions with exponentially distributed firing times, and black filled rectangles depict transitions with a constant delay unequal to zero. Transitions are joined to places by input arcs, output arcs or inhibitor arcs. The latter forbids firing as long as the corresponding place is marked. Priorities can be attached to immediate transitions to resolve conflicts. Places can have a capacity of more than one token and arcs can have a multiplicity of greater than one. The underlying time scale is continuous.

Analysis of DSPNs has been described in [6] [9], and several tools are available. We have carried out experiments that require manual translation, using TimeNET [23] but we are working on an automated integration of both tools. An essential step towards this integration is the development and implementation of an SEFT-to-DSPN translation algorithm, which will be introduced in section 3.4.

The translation of SEFT states and events to DSPN places and transitions is straightforward: each state is mapped to a place and each event to a transition. SEFT gates are translated as a whole by looking up the corresponding DSPN structure in a dictionary. The gate dictionary can be found in the Appendix. Dashed places or transitions signify import places/transitions, which are references to places/transitions (marked as “export”) in other component DSPNs. During flattening (integration of the partial nets), the import elements will be merged with the corresponding export output. The semantics of the gates is best understood by playing the “token game”. For instance, the export place $P_{out}$ of the AND (State x State) gate net is marked if and only if someone puts tokens to both import Places $P_{in1}$ and $P_{in2}$. The situation is different with the Priority AND (Event x Event) gate: here, the leftmost input transition must fire first, then the middle input transition and after that the right input transition to make the output transition fire.

Our semantics, which lies in the composition by state ports and event ports, cannot be translated directly to a composition in the Petri net domain, so special patterns are necessary when translating SEFT ports to DSPNs. The reasons are that ports must not introduce any backward influences and events are not stored as Petri net tokens are.

---

Since some logical gates can have an arbitrary number of inputs, their DSPN equivalent will not just be “looked up”, but dynamically be generated by the dictionary component according to the actual number of input ports. For this purpose we have partitioned the set of DSPN nodes into parts that remain unchanged and partial nets, which we call “import building blocks”, corresponding to those input ports whose number can vary. In the appendix, building blocks are indicated by a dashed border and can be inserted multiple times into the DSPN gate structure, connecting them (with the exception of Priority gates) to the same places and transitions of the immutable part as their peers, using the same kind of arcs. In the case of Priority gates the connection principle is a little different: because their building blocks also communicate with each other they are “chained”, which is indicated by dashed arcs in the DSPN. If the Priority gate has only two inputs (three, in the presence of a reset input) the building block shown is left out altogether, replacing it with a standard arc.
In order to translate an event port, an additional place is added where the triggering transition puts a token when firing. Triggering only occurs if the triggered transition is feasible at the same instant of time because there is no storage of events. To capture this, we add an artificial immediate transition with lower priority than the triggered transition so that it consumes the token in those cases when it is not immediately used. The event port (or trigger) pattern can be seen in Fig. 3-4, before and after flattening. Note that in DSPNs priority 2 takes precedence over priority 1.

An important point regarding state ports is that the target component must not backward modify the source component state (and thus the DSPN marking). This can be achieved by a pair of anti-parallel edges between the (guarding) state and the (guarded) event. Whenever a token is consumed, it is immediately put back.

3.4 The Translation Algorithm

The translation and its preparation comprises the following steps3:

1st Step – Check of preconditions

- The component model to be analysed and all nested subcomponent models (recursively) are available and are valid SEFT models
- The component nesting hierarchy is free from cycles (a component must not refer directly or indirectly to itself as a subcomponent)
- All component input ports are connected on the higher hierarchy level
- All causal paths are free from untimed cycles (this must be checked across component borders and hierarchy levels)
- The interface (“footprint”) of every subcomponent instance matches the actual component model for that subcomponent

2nd Step – Model-wise translation (create target DSPN for each model)

Create new (global) “correspondence table” to store the IDs of SEFT component models and their corresponding DSPN translations in

For each SEFT component model (following the component nesting hierarchy of the top model bottom-up)

- Create new (local) “crossreference table” to store the IDs of SEFT nodes and of their DSPN counterparts in
- Represent each state by a place in target DSPN
  - If the state succeeds the init event, mark the place with a token (The init event itself has no DSPN counterpart)
  - Put the IDs of state and place into crossreference table
- Represent each deterministic or exponential event by a deterministic or exponential DSPN transition and transfer the delay (or rate) parameter. Add IDs to crossreference table

3 For sake of clarity, details of the translation process, such as adequate positioning of nodes and the transfer of other non-structural net properties, are omitted.
Represent each triggered event by an immediate transition, except the init event. Add IDs to crossreference table.

For each in- or outport of the component itself:
- If it is an event port, add an immediate transition to the target DSPN. Add IDs to crossreference table and mark entry as “export”
- If it is a state port, add a place to the target DSPN. Add IDs to crossreference table and mark entry as “export”

For each subcomponent instance, insert its footprint into the target DSPN,
- For each event port of the subcomponent node, insert an immediate transition into the target DSPN, prefixing IDs with the ID of the subcomponent instance. Add IDs to the crossreference table and mark entry as “import”
- For each state port of the subcomponent node, insert a place into the target DSPN, prefixing IDs with the ID of the subcomponent instance. Add IDs to the crossreference table and mark entry as “import”

For each logical gate, get corresponding DSPN structure from the gate dictionary and insert it into the target DSPN. Add IDs of its ports and their DSPN counterparts to crossreference table.

For each temporal edge, except edges starting in the init event, which have no DSPN counterpart, add a standard arc to the target DSPN. Connect starting and end point of arc to the DSPN nodes that correspond to source and target of the SEFT edge, using the crossreference table.

For each causal edge:
- If the edge leads from an event (or an event port of the component model itself, of a gate or of a subcomponent instance) to an event (“trigger relation”)
  - Add an instance of the trigger pattern to the target DSPN
  - Merge the trigger’s source and target transitions with those transitions of the target DSPN that correspond to source and target of the SEFT edge and replace them with a new transition. Change entries in crossreference table accordingly
- If the edge leads from a state (or a state port of the component model itself, of a gate or of a subcomponent instance) to an event (“guard relation”)
  - Add a pair of standard arcs to the target DSPN
  - Set the source of the first and the target of the second arc to the transition in the target DSPN that corresponds to the target of the SEFT edge, using the crossreference table
  - Set the target of the first and the source of the second arc to the place in the target DSPN that corresponds to the

---

4 If the subcomponent’s SEFT model has already been translated (i.e. the supercomponent contains more than one instance of it), just insert a copy of all exported elements of the translated subcomponent into target DSPN the same way.
source of the edge in the SEFT component model, using the crossreference table

- If the edge leads from a state or state port to a state port or from an event or event port to an event port, merge those nodes in the target DSPN that correspond to source and target of the SEFT edge\(^5\) and replace them with a new node. Change entries in crossreference table accordingly
  - Copy all entries marked “export” or “import” from crossreference table to a new table, called “interface table”, and add table to target DSPN model
  - Add IDs of SEFT component model and its DSPN translation to correspondence table

3\(^{rd}\) Step – Flattening (create a single DSPN out of the DSPNs just translated)

For each SEFT component model that contains subcomponent instances (following the component nesting hierarchy of the top component model bottom-up)

- For each subcomponent instance
  - Using the correspondence table from the first translation step, insert a copy of the DSPN corresponding to the subcomponent model into the DSPN that corresponds to the (enclosing) component model, prefixing all elements of the DSPN subcomponent model with the ID of the SEFT subcomponent instance to avoid name conflicts
  - Prefix DSPN element IDs in the subcomponent model’s interface table with the ID of the subcomponent instance in the enclosing component model and copy entries to interface table of the enclosing component model
  - For each import entry in the enclosing component model’s interface table
    - Identify DSPN node node\(_1\)
    - Find exported counterpart node\(_2\) in the interface table
    - Merge node\(_1\) and node\(_2\), replacing them with a new node
    - Delete those two entries from interface table that correspond to merged nodes and replace every remaining occurrence of ID of node\(_1\) or node\(_2\) with ID of the new node

One exported place or transition can be merged with several counterparts. It is a failure if port references cannot be resolved or if import places or transitions have no export counterpart. After the top-level SEFT has been translated, the place or transition connected to the output port selected for analysis must be identified. We intend to introduce a net simplification step before the flattening steps on each hierarchy level to reduce the state space.

---

\(^5\) In merging DSPN elements, special considerations in the case of conflicting properties apply (e.g. an exponential transition wins over an immediate transition).
3.5 Performing the Analysis

For analysis, the requested measure (e.g. average probability of a state term that is connected to an output port) is translated into a measure that can be calculated by the DSPN analysis tool (e.g. the marking probability for a place that corresponds to the system state of interest). Then a suitable analysis procedure has to be started. The TimeNET tool we are currently using offers both transient and steady-state analysis for DSPNs, as well as simulation. Analysis is faster, but due to the analysis algorithm used it can only be applied in cases where at most one deterministic transition is enabled in any marking. If this condition is violated, simulation is still possible. At present state, we read back the results of the analysis manually. Our tool ESSaRel that is currently under development will start the analyser via API or command line call and later read the calculated values from the TimeNET result file to display them in its own GUI.

4 The Fire Alarm System

4.1 Description of the Example System

In this section we demonstrate the presented approach by the example of a fire alarm system. The system consists of two redundant fire alarm units which may fail stochastically. The hazard to be analysed is the situation when both alarm units are simultaneously in the “failed” state, since in this case a fire might break out without being noticed. A watchdog periodically checks the alarm units and restarts them if they are in failed state.

4.2 Constructing the SEFT

We start by modelling each of the units as independent components. The alarm units are instances of the component “fire alarm unit” shown in Fig. 4-1.

An alarm unit may be running properly or fail stochastically with a failure rate of \( \lambda = 1/10 \) hour\(^{-1}\). In order to restart a unit an external trigger is needed. Before normal
operation, some initialization steps taking a deterministic time of 0.1 hours have to be performed. In order to notice when a unit is out of work, a state output port (depicted by the filled S triangle) that senses if the unit is running is used in the model. For external triggering of the initialization routine an event input port (the empty E triangle) is needed. The watchdog, as shown in Fig. 4-1 b), is simply depicted as a component with only one state. The triggering event is produced once every hour and can be connected to other components via an event output port.

Now that the technical units are given, we combine the modelled units to form the SEFT that describes the complete fire alarm system and explains the causal paths that lead to the hazard situation. Fig. 4-2 shows how the fire alarm system is modelled, using two instances of the “fire alarm unit” and one instance of the “watchdog” component. The inner structure of the instances is omitted in this view. The watchdog is connected to the event input ports of both alarm units so that it can trigger a restart of a failed unit when necessary. Since the fire hazard is present when both of the redundant alarm units are not working at the same time they are combined with a NOT gate each, which in turn serve as inputs for a state AND gate. The output of the state AND gate represents the hazard situation.

All preconditions for analyzing this model are fulfilled: There are no cycles in the component hierarchy or in any causal relation and all state and event port have been connected to their counterparts. The state output port of the AND gate does not need to be connected since it represents the hazard situation to be analyzed.

4.3 Translation into DSPN

Before performing the analysis, the model has to be translated into a DSPN. The “alarm unit” and “watchdog” components can be translated step by step. Each state is mapped onto a DSPN place, the events that occur stochastically onto exponential transitions, the events occurring after a certain time onto deterministic transitions and externally triggered events onto immediate transitions, respectively. The resulting parts have to be combined with the gate nets to form the complete flattened DSPN shown in Fig. 4-3. The dotted rectangles show the DSPN parts representing the components. The markings were set in such a way that the initial state of the alarm system is conceived: the alarm units are working properly and there is no fire hazard. Note
the trigger pattern applied in combining the watchdog and the externally triggered events in the alarm units.

4.4 Analysis of the Flattened DSPN

The DSPN can now be examined using TimeNET. This step as well as the translation into DSPNs must currently be done manually but an export filter to the TimeNET file format will be integrated in our tool ESSaRel. The hazard situation is represented by the place marked with $P_{\text{out}}$ which was mapped on the output of the state AND gate. The probability that one token lies in place $P_{\text{out}}$ denotes the probability that both alarm units are out of work simultaneously. The corresponding expression in TimeNET is "$P\{\#P_{\text{out}}=1\}$".

Fig. 4-3: DSPN of the fire alarm system after flattening

TimeNET cannot accomplish an analysis of the net, since more than one deterministic transition might be enabled at the same time. A steady-state simulation (continuous time) with the parameters given in Fig. 4-1 and a maximum relative error of 1% (a parameter TimeNET uses to specify the desired accuracy of the simulation)
returns a resulting probability for the fire hazard as 0.003975. In other words, in the long run in about 0.4% of the operation time both alarm units are not working so that a fire hazard persists.

5 The ESSaRel Tool Project

In 2003, the ESSaRel project (Embedded Systems Safety and Reliability Analyser) was created with the purpose to create a platform for the integration of different modelling techniques [7]. To support this goal, a comprehensive tool was needed that allows to handle different kinds of models (and even the later extension to new modelling techniques), to analyse models with appropriate algorithms and to translate models into other models. Due to their expressive power that unites features from different safety and reliability modelling techniques, SEFTs were chosen as the central model within the ESSaRel project.

![ESSaRel screenshot, showing a Markov chain and an SEFT model](image)

A modeling project within ESSaRel can spread across multiple files and is organized by components, where each component can contain one or more models. For each model, specific analyses or translations can be defined. Analyses perform mathematical operations on a model, producing a list of qualitative or quantitative results (e.g. a list of prime implicants for a fault tree, a list of steady-state probabilities for the states of a Markov chain). Internally, these analysis algorithms refer to different translations that transform ESSaRel models into other models (e.g. state charts...
into SEFTs, hierarchical SEFTs into flat SEFTs). For each of the included models, different analyses and translations are possible that are specified in an XML configuration file. In this way, the step-wise translation, flattening and evaluation of SEFTs as described in section 3.4 could be implemented in a clear layout. The ESSaRel tool uses an extensible XML format to store the models but also offers the possibility to attach import and export filters to foreign file formats. This way, the generated DSPNs are exported to TimeNET and analysis results can be read back from the TimeNET output. An import interface to the CASE-tool Rational Rose RT that uses a variant of ROOMcharts has also been implemented in order to allow model reuse from the design phase. A screenshot of ESSaRel at work is shown in Fig. 5-1.

6 Conclusion and Further Research

We have proposed State-Event-Fault Trees as a new modelling technique for safety and reliability analysis that unites elements from Fault Tree Analysis and from Statecharts. This technique extends the modelling capabilities of FTs for embedded systems and enables us to integrate finite state models with FTs. In particular we can integrate Markov chains and Statecharts, allowing the reuse of design models for safety analysis. We explicitly distinguish states and events and therefore extend the traditional set of FT gates by the introduction of typed gates. Phenomena like temporal order of events that were not expressible by FTs before can be expressed by SEFTs, and gates with memory such as History AND or Priority AND can be defined formally. We have developed new kinds of gates to match frequent structures of safety-critical systems, such as History/Priority AND with a Reset input (for repair/restart situations) or History/Priority AND with a time parameter that indicates how close both events must occur to each other. The component concept that has been borrowed from Component Fault Trees allows partitioning SEFTs in accordance with the actual technical component structure and facilitates the handling of complex projects. In summary, the established top-down causal analysis search provided by FTA can now be combined with a powerful and reusable description of embedded system behaviour brought in by the integration of state-based models.

For quantitative probabilistic analysis of SEFTs we translate them component-wise into DSPNs, apply simplifications on component level and merge them to one flat net. Existing Petri net analysis tools, for instance TimeNET, calculate the measures that correspond to the hazard probabilities in the original SEFT. We have shown the applicability of our method by the example of a fire alarm system. We carried out other small studies to validate the model; larger studies with industrial partners are under preparation.

We work towards a closer integration of state-based and combinatorial analysis approaches to obtain better analysis performance for subcomponents that can be described by a combinatorial model, similar to the approaches given in [12]. We also plan to do some simplification on component level before integration, exploiting the fact that only a few of the functional states of each component are relevant for safety or reliability considerations.
The ongoing integration of SEFTs into a usable tool is part of our research project ESSaRel. The platform for this integration is the existing CFT tool UWG3 which has been developed in cooperation with Siemens. We have defined the translation and flattening algorithm and are now implementing the export filter to TimeNET and an import filter to read models from the CASE tool RationalRose RT that are mainly based on ROOM charts. In the end we are working towards an integrated tool chain for safety and reliability analysis of embedded systems.

7 References


Appendix – The Gate Dictionary

<table>
<thead>
<tr>
<th>Gate name &amp; SEFT graphical representation</th>
<th>DSPN equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND_State(n)</td>
<td><img src="image" alt="DSPN equivalent" /></td>
</tr>
</tbody>
</table>
EQUAL(n)

History_AND(n)
State-Event Fault Trees - A Safety Analysis Model for Software Controlled Systems

Priority\_AND(n, t)

Priority\_AND\_R(n)

Det\_Delay(t)

Prob\_Delay(t)
UPON

ENTER

LEAVE

UNTIL