Combining Real-Time Model-Checking and Fault Tree Analysis *

Andreas Schäfer

Department of Computing Science, University of Oldenburg, 26111 Oldenburg, Germany

schaefer@informatik.uni-oldenburg.de

Abstract. We present a semantics for fault tree analysis, a technique used for the analysis of safety critical systems, in the real-time interval logic Duration Calculus with Liveness and show how properties of fault trees can be checked automatically. We apply this technique in two examples and show how it can be connected to other verification techniques.

Keywords: Real-time systems, model-checking, fault tree analysis

1 Introduction

In this paper we bring together the two worlds of safety engineering on the one hand and real-time model-checking on the other hand. We present an approach of using model-checking to determine whether a fault tree is designed properly. Fault tree analysis [VGRH81] is a technique widely used by engineers to analyse safety of safety-critical systems. Originally, it did not have a formal semantics and relied on the expertise of safety engineers. Recently there have been several attempts to define a formal semantics for fault trees [RST00,Han96]. In this paper we go one step further and show how to combine the fault tree analysis with real-time model-checking.

Both parties benefit from this combination. From the point of view of the safety engineer formal models and proofs by model-checking raise the quality of safety analysis. The aim is to make implicit assumptions on the behaviour of the system explicit and to discover problems that have been overlooked. So we add extra redundancy to the safety analysis itself.

On the other hand, model-checking benefits because the formal model is compared with the fault tree that is created from the system independently. Additionally, the knowledge of the system which is present in the fault tree can be used to simplify the verification process. Instead of verifying one complex property of the whole system, we decompose the property into simpler properties of subsystems using fault tree analysis. Then we verify that the decomposition is correct and finally show that the simple properties hold.

As the underlying formalism we use Duration Calculus with Liveness (DCL) [Ska94], which is designed to describe and reason about real-time systems. As the

* This research was partially supported by the DFG under grant Ol/98-2.

K. Araki, S. Gnesi and D. Mandrioli (Eds.): FME 2003, LNCS 2805, pp. 522-541, 2003.

operational formalism for model-checking we use Phase Automata [Tap01,DT03] because they have a semantics in Duration Calculus. Since we define the fault tree semantics in Duration Calculus with Liveness, too, we completely stay in this formal framework.

As an example consider the fault tree in figure 1. Let it be designed for a system in which a relay K2 controls a pump which pressurizes a tank. We assume that the tank will burst if the contacts of relay K2 are closed for more than 60 seconds. The fault tree decomposes this event and states that if it occurs then either an electromagnetic field (EMF) must have been applied to the coil for more than 60 seconds or erroneously the relay does not open. The aim is to verify that this is in fact true. To this end, we create an operational model of our relay to express our assumptions on its behaviour. We formalise each event given in the fault tree by a formula in Duration Calculus with Liveness. In this example let E_1, E_2 and E_3 be these formalisations. We can be sure that no cause of the event E_1 is forgotten if the implication $E_1 \Rightarrow (E_2 \vee E_3)$ holds with respect to our operational model $\mathcal{M}$ of the relay. So in fact we have to verify

$$(\mathcal{M} \wedge E_1) \Rightarrow (E_2 \vee E_3).$$

This is done by translating each formula and its complement into a Phase Automaton. Let these Phase Automata be called $\mathcal{A}_{E1}, \mathcal{A}_{E2}, \mathcal{A}_{E3}$ and $\mathcal{A}_{\neg E1}$, $\mathcal{A}_{\neg E2}$, $\mathcal{A}_{\neg E3}$. We check whether there is a run of the model $\mathcal{M}$ which is also possible for $\mathcal{A}_{E1}$, $\mathcal{A}_{\neg E2}$ and $\mathcal{A}_{\neg E3}$. If this is not the case, the implication is true. Thus no causes of the event E_1 have been overlooked.

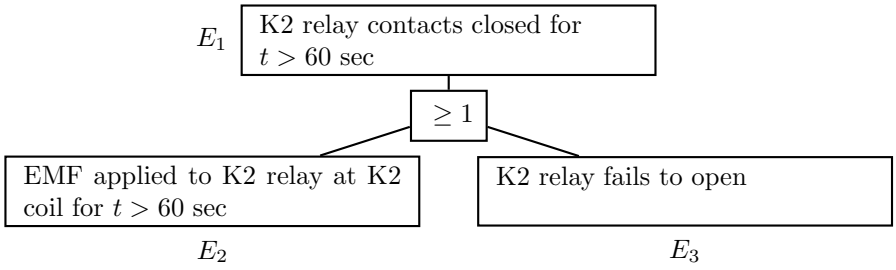


Fig. 1. Example of decomposition

In this paper we give precise semantics of fault trees to express that a fault tree is well designed. Apart from the or-connective considered in this example the other connectives that may appear in fault trees are also treated. For a subclass of DCL formulae which is relevant for fault trees we give algorithmic constructions of Phase Automata. And we show how they can be composed so we can use model-checking to establish that the fault tree is well designed for a given model of the system.

The rest of this paper is organised as follows. In section 2 and 3, we introduce Duration Calculus with Liveness and Phase Automata. In section 4 we give a semantics for fault trees in Duration Calculus with Liveness. In section 5 we show how properties can be model-checked automatically using Phase Automata. This

approach is applied to one example in section 6 and one case study in section 7 where we design and verify a more complex system. We integrate the fault tree analysis into a verification process with PLC-Automata [Die00] which can be directly compiled into software for embedded systems and into timed automata.

2 Duration Calculus with Liveness

Duration Calculus (DC for short) [ZHR91] is a real-time interval logic which allows reasoning about durations of states. As the properties which will be important for fault trees will be liveness properties, we use the extension *Duration Calculus with Liveness* (DCL) [Ska94], which introduces special modalities to express real liveness properties.

Real-time systems are described by a finite number of *observables* (time-dependent variables) which are denoted by $X, Y, \ldots$ and interpreted by an interpretation $\mathcal{I}$ which assigns to each observable X a function $\mathcal{I}(X) : \mathsf{Time} \to D$. Here Time is the time domain – in this case the real numbers – and D is the finite domain of the observable. Additionally we use rigid variables denoted by $x, y, \ldots$ and valuations $\mathcal{V}$ which assign a real number to each rigid variable.
State assertions π are generated by the grammar

$$\pi ::= 0 \mid 1 \mid X = c \mid \neg\pi_1 \mid \pi_1 \wedge \pi_2$$

and describe the state of the real-time system at a certain point of time, with the semantics:

$$\mathcal{I}[\![0]\!](t) = 0, \quad \mathcal{I}[\![1]\!](t) = 1, \quad \mathcal{I}[\![X = k]\!](t) = \begin{cases} 1 & \text{if } \mathcal{I}(X)(t) = k \\ 0 & \text{otherwise} \end{cases}$$

and the usual definition for the propositional connectives. Duration terms θ are either rigid variables or derived from state assertions using the $\int$ operator; their semantics depends on an interpretation $\mathcal{I}$, a valuation $\mathcal{V}$, and an interval $[a, b]$, and is defined by

$$\mathcal{I}[\![x]\!](\mathcal{V}, [a, b]) = \mathcal{V}(x), \quad \mathcal{I}[\![\textstyle\int P]\!](\mathcal{V}, [a, b]) = \int_a^b \mathcal{I}[\![P]\!](t)dt$$

Duration formulae F are generated by the grammar

$$F ::= p(\theta_1, \ldots, \theta_n) \mid F_1; F_2 \mid F_1 \lhd F_2 \mid F_1 \rhd F_2 \mid \neg F_1 \mid F_1 \wedge F_2 \mid \exists x F \mid \exists X F$$

and are evaluated in a given interpretation $\mathcal{I}$, a valuation $\mathcal{V}$, and a time interval $[a, b]$. The symbol p denotes a predicate symbol like $=, \leq, \geq$. In general, the meaning of a predicate p is given by the interpretation and denoted by $p_{\mathcal{I}}$. A formula $F_1; F_2$ holds iff the given interval can be "chopped" into two parts such that F_1 holds on the left part and F_2 on the right part. The expanding modalities $\lhd$ and $\rhd$ allow an expansion of the interval to the left respectively to the right. Additionally to negation and conjunction we allow quantification over

rigid variables and observables. Other propositional connectives can be defined as abbreviations. Formally,

$$\begin{aligned}
&\mathcal{I},\mathcal{V},[a,b] \models p(\theta_1,\ldots,\theta_n) && \text{iff } p_{\mathcal{I}}(\mathcal{I}[\![\theta_1]\!](\mathcal{V},[a,b]),\ldots,\mathcal{I}[\![\theta_n]\!](\mathcal{V},[a,b]))\\
&\mathcal{I},\mathcal{V},[a,b] \models F_1;F_2 && \text{iff } \exists\, k \in [a,b] : \mathcal{I},\mathcal{V},[a,k] \models F_1 \text{ and } \mathcal{I},\mathcal{V},[k,b] \models F_2\\
&\mathcal{I},\mathcal{V},[a,b] \models F_1 \rhd F_2 && \text{iff } \exists\, k \geq b : \mathcal{I},\mathcal{V},[a,k] \models F_1 \text{ and } \mathcal{I},\mathcal{V},[b,k] \models F_2\\
&\mathcal{I},\mathcal{V},[a,b] \models F_1 \lhd F_2 && \text{iff } \exists\, k \leq a : \mathcal{I},\mathcal{V},[k,a] \models F_1 \text{ and } \mathcal{I},\mathcal{V},[k,b] \models F_2
\end{aligned}$$

The definitions of the remaining connectives and quantifications over rigid variables and observables are like in first-order logic. Additionally, the following abbreviations will be used:

$$\begin{aligned}
&\ell && \stackrel{df}{=} \textstyle\int 1 \text{ (length of the interval)} && \lceil P \rceil && \stackrel{df}{=} \textstyle\int P = \ell \wedge \ell > 0\\
&\Diamond F && \stackrel{df}{=} (true; F; true) \text{ (somewhere)} && \Box F && \stackrel{df}{=} \neg\Diamond\neg F\\
&\Diamond_L F && \stackrel{df}{=} (true; F; true) \rhd true \text{ (eventually)} && \Box_L F && \stackrel{df}{=} \neg\Diamond_L\neg F \text{ (always)}
\end{aligned}$$

3 Phase Automata

As an operational model for real-time systems we use Phase Automata [Tap01], which possess a formal semantics in DC and allow model-checking using the tool Moby/DC [DT03] . The intuition is similar to Timed Automata [AD94].

A Phase Automaton $\mathcal{A} = (P, E, C, cl, s, d, P_0)$ consists of finite sets of states P, and clocks C, a transition relation $E \subseteq P \times P$, and a set P_0 of initial states. The function cl assigns a set of clocks to each state, the function s assigns a state assertion to each state, and the function d assigns to each clock a time interval.

A Phase Automaton can stay in the present state only if the state assertion holds. Additionally, for each clock c the amount of time the automaton stays in states in $cl^{-1}(c)$ must be within the interval given by $d(c)$.

In figure 2 we present an example of a Phase Automaton modelling the formula $\Diamond_L(\lceil P \rceil \wedge 4 < \ell; \lceil Q \rceil \wedge 3 < \ell)$. The open intervals $(0,\infty)$ and $(4,\infty)$ express that the automaton may stay in s_0 and s_1 arbitrarily long but has to leave these states eventually, whereas the interval $(3,\infty]$ allows the automaton to stay in s_2 forever.

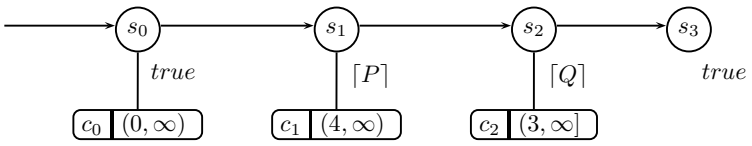


Fig. 2. Phase Automaton for $\Diamond_L(\lceil P \rceil \wedge 4 < \ell; \lceil Q \rceil \wedge 3 < \ell)$

3.1 Semantics

The semantics of a Phase Automaton $\mathcal{A}$ is defined in terms of one big DC formula. It encodes the behaviour using one fresh observable $\mathsf{ph}_{\mathcal{A}}$, which ranges over the state space of the automaton. The subformulae model the initial states, the successor state relation, and the clock constraints. To give a flavour of these formulae we just present one of them. It expresses that it is impossible to stay in a set of states which belong to the same clock c longer than the upper bound given in the clock interval $d(c)$. It encodes the progress of the automaton.

$$D^c_{pr}(\mathcal{A}) \stackrel{df}{=} \neg\Diamond(\lceil \bigvee_{p \in cl^{-1}(c)} \mathsf{ph}_{\mathcal{A}} = p \rceil \wedge \ell > d(c))$$

3.2 Model-Checking and closure under complementation

The model-checker Moby/DC [Tap01,DT03] checks whether a set of Phase Automata running in parallel have a common run. To exploit this, we use the following automata-theoretic approach to model-checking. We model the system to be checked by a set of Phase Automata. The property which is to be verified is negated. For this negated property we also construct a Phase Automaton and check whether there is a run which satisfies both the model of our system and the negated property. If this is not the case, the property holds. Unfortunately, Phase Automata are – like Timed Automata – not closed under complementation. Therefore we will have to restrict ourselves to a subset of Phase Automata that permits complementation.

4 Fault Tree Analysis

Fault tree analysis (FTA) [VGRH81] is an engineering technique to identify causes of system failures. Its main area of application are safety critical components in nuclear and aviation industries. Starting with an undesired event (called top-event) all possible causes (called sub-events) are identified. These causes are joined using and and or gates to the top-event depending on whether all events have to occur to yield the top-event or whether one event is sufficient. This procedure is iterated until a given granularity is reached. Events that are not developed further are called basic-events. In figure 1 we gave an example taken from the Fault Tree Handbook [VGRH81] in which for one event two possible causes are identified. We use the notation defined by the IEC 61025 standard [IEC93].

4.1 DCL Semantics

In order to use model-checking techniques to verify that a fault tree is constructed properly and to combine it with other formal techniques in one verification process we need a formal semantics. Originally, there was no formal semantics [VGRH81] but there have been several attempts [Han96,BA93,RST00] to define one in order to avoid ambiguities.

Events. Events are formalised by DCL formulae. Gorski [Gór94] divides the events occurring in fault trees into three groups. So we will restrict ourselves to DCL formulae for these groups and give a DCL formula pattern for each of them. We require the events to be formalized by such a DCL formula.

$$\Diamond_L \lceil \pi \rceil \wedge a \sim \ell \qquad \text{(Reachable state)}$$

$$\Diamond_L \Box_L \lceil \pi \rceil \wedge \neg \Diamond_L (\lceil \pi \rceil ; \lceil \neg \pi \rceil) \qquad \text{(Final state/Deadlock)}$$

$$\Diamond_L (\lceil \pi_1 \rceil \wedge a_1 \sim \ell ; \lceil \pi_2 \rceil \wedge a_2 \sim \ell \sim b_2 ; \ldots ; \quad \lceil \pi_{n-1} \rceil \wedge a_{n-1} \sim \ell \sim b_{n-1} ; \lceil \pi_n \rceil \wedge a_n \sim \ell) \qquad \text{(State sequence)}$$

where $\sim \in \{<, \leq\}$, a and b are real numbers and π is a state assertion, describing the system state. Upper bounds for the duration are to be specified using the sequence pattern like $\Diamond_L (\lceil \neg \pi \rceil ; \lceil \pi \rceil \wedge \ell \leq b ; \lceil \neg \pi \rceil)$. Examples for these type of pattern could be:

- Reachable state: The relay is eventually closed for more than 60 sec.
- Deadlock: Once the relay is closed, it will never open again.
- State sequence: Eventually the relay is closed for at least 60 sec and is open for at least 20 sec after that.

Gates. Events are joined by gates to express their dependence. We follow Reif et al.'s approach [RST00] and distinguish two kinds of gates:

- *decomposition gates* which do not impose any temporal relationship between the events joined by this gate, and
- *cause consequence gates* which express temporal dependencies.

We consider the two decomposition gates and ($\wedge$) and or ($\vee$) and three cause consequence gates. For the asynchronous or ($\vee$-acc) gate we require the event to happen after *one* of the causes have happened. For the asynchronous and gate ($\wedge$-acc) we require the event to happen after *all* of the causes have happened. And for the synchronous and gate ($\wedge$-scc) we require the event to happen after *all* events occurred *simultaneously*, which means that there is a time interval in which all formulae expressing the event hold.

After having given ideas what events and gates are, we can proceed to define syntax and semantics for fault trees.

4.2 Syntax

The set of *fault trees* is defined inductively as follows.

- Every basic event E formalized as a DCL formula is a fault tree with top event E.
- For a nonempty and finite set $\mathcal{T}$ of fault trees, a gate $G \in \{\wedge, \vee, \wedge\text{-scc}, \wedge\text{-acc}, \vee\text{-acc}\}$ and an event E, the term $(E, G, \mathcal{T})$ is a fault tree with top event E. As all gate conditions will be associative we do not have to impose an order on the fault trees in $\mathcal{T}$.

We continue to present fault trees graphically and not as a term structure. According to the IEC 61025 standard [IEC93] we use "&" and "≥ 1" in the graphical notation instead of "$\wedge$" and "$\vee$" and omit and gates with one child.

4.3 Semantics

For each gate we define two DCL proof obligations, one stating that the occurrence of sub-events is necessary for the event to occur and the other that they are sufficient. For fault tree T, we define in our semantics $[\![T]\!]_S$ to be the conjunction of all sufficient conditions and $[\![T]\!]_N$ as the conjunction of all necessary conditions. The proof obligations for the necessary conditions are especially important. They express that no cause has been overlooked. If all necessary conditions can be proved by model-checking, we call a fault tree *complete*. Additionally, if it can be proved that the sub-events cannot happen, it follows that the event itself cannot happen.
The precise semantics is defined inductively:

- For every basic event E we define $[\![E]\!]_N \stackrel{df}{=} [\![E]\!]_S \stackrel{df}{=} \mathsf{true}$
- Let $T = (E, G, \{T_1, \ldots, T_n\})$ be a fault tree and let $E_1, \ldots, E_n$ be the top events of $T_1, \ldots, T_n$. Then we define

$$[\![T]\!]_S \stackrel{df}{=} F_S^G \wedge \bigwedge_{i=1}^{n} [\![T_i]\!]_S$$
$$[\![T]\!]_N \stackrel{df}{=} F_N^G \wedge \bigwedge_{i=1}^{n} [\![T_i]\!]_N$$

  where the proof obligations F_S^G and F_N^G are given by

$$F_S^{\wedge} \stackrel{df}{=} \left(\bigwedge_{i=1}^{n} E_i \right) \Rightarrow E \qquad (\wedge\text{-gate: sufficient condition})$$
$$F_N^{\wedge} \stackrel{df}{=} E \Rightarrow \left(\bigwedge_{i=1}^{n} E_i \right) \qquad (\wedge\text{-gate: necessary condition})$$
$$F_S^{\vee} \stackrel{df}{=} \left(\bigvee_{i=1}^{n} E_i \right) \Rightarrow E \qquad (\vee\text{-gate: sufficient condition})$$
$$F_N^{\vee} \stackrel{df}{=} E \Rightarrow \left(\bigvee_{i=1}^{n} E_i \right) \qquad (\vee\text{-gate: necessary condition})$$

For the cause consequence gates we have to express the notion of "after" in terms of DCL. So we have to get rid of the expanding modalities in $\Diamond_L$ and use the substitution $\{\Diamond_L/\}$ which removes the leading occurrence of a $\Diamond_L$ operator from a formula. After this substitution the formula is still a well-formed DCL formula.

$$F_S^{\wedge\text{-acc}} \stackrel{df}{=} \left(\bigwedge_{i=1}^{n} E_i \right) \Rightarrow E \qquad (\wedge\text{-acc-gate: sufficient condition})$$
$$F_N^{\wedge\text{-acc}} \stackrel{df}{=} \neg\left((\neg \bigwedge_{i=1}^{n} \Diamond E_i\{\Diamond_L/\}); E\{\Diamond_L/\} \right) \qquad (\wedge\text{-acc-gate necessary condition})$$

$$F_S^{\vee\text{-acc}} \stackrel{df}{=} \left(\bigvee_{i=1}^{n} E_i \right) \Rightarrow E \qquad (\vee\text{-acc-gate: sufficient condition})$$

$$F_N^{\vee\text{-acc}} \stackrel{df}{=} \neg\left((\neg \bigvee_{i=1}^{n} \Diamond E_i\{\Diamond_L/\}); E\{\Diamond_L/\} \right) \qquad (\vee\text{-acc-gate: necessary condition})$$

$$F_S^{\wedge\text{-scc}} \stackrel{df}{=} \Diamond_L \left(\bigwedge_{i=1}^{n} E_i\{\Diamond_L/\} \right) \Rightarrow E \qquad (\wedge\text{-scc-gate: sufficient condition})$$

$$F_N^{\wedge\text{-scc}} \stackrel{df}{=} \neg\left((\neg\Diamond \bigwedge_{i=1}^{n} E_i\{\Diamond_L/\}); E\{\Diamond_L/\} \right) \qquad (\wedge\text{-scc-gate: necessary condition})$$

If the consequence has already happened due to other causes the consequence does not need to happen again. So for the sufficient condition we do not require the consequence to happen after the causes.

5 Model-Checking

The proof obligations presented in the previous section should be verified automatically by model-checking. As one can only prove correctness propositions with respect to a given model, we assume that a formal model of the system is given in terms of Phase Automata.
So the model-checking problem for a given fault tree T is to check if a model $\mathcal{M}$ satisfies

$$\models \mathcal{M} \Rightarrow [\![T]\!]_N$$
$$\models \mathcal{M} \Rightarrow [\![T]\!]_S.$$

Zhou et al. [ZHS93] showed that the Duration Calculus is undecidable. As we can write every DC Formula F as $\mathsf{true} \Rightarrow F$ and this is the necessary condition of the fault tree $(true, \wedge, (F))$ the model-checking problem for fault trees is undecidable in general. So a restriction to a subset of DCL is unavoidable. Therefore we only consider the three classes of events proposed by Gorsky [Gór94] and restrict the formulae to patterns given in section 4.1.

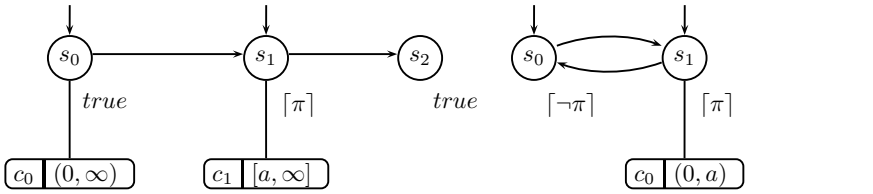
If we assume for the third pattern that two different state expressions occurring in a DCL formula of that type cannot hold at the same time, all formulae matching one of these patterns can be translated into Phase Automata. Therefore the fragment is decidable. The expressiveness can only be evaluated by considering case studies which we do in sections 6 and 7.

Idea. To verify the gate conditions arising from fault tree analysis they are translated into Phase Automata, too. Model-checking establishes whether or not the conditions hold for the given model. First we present constructions of Phase

Automata for each event pattern of section 4.1 and its complement. After that we show how the proof obligations arising from different gates can be model-checked.

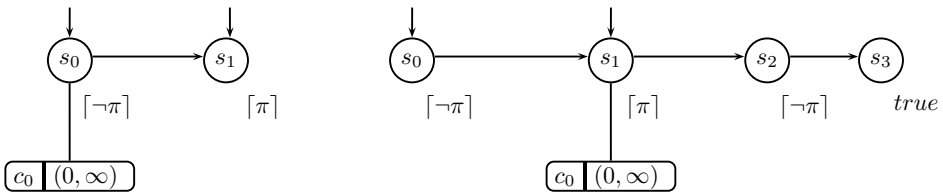
5.1 Constructing Phase Automata for Event Patterns

The construction of Phase Automata for most of the event patterns of section 4.1 is given in figures 3 to 5. Only the construction of the complement automaton for the sequence pattern requires extra attention. For all constructions given in this section, we assume that "$\leq$" is the only relation occurring in the DCL formula. If this is not the case, the corresponding open intervals have to become closed and vice versa.



(1st Pattern) (Complement of 1st Pattern)

Fig. 3. Phase Automata for the first pattern of fault tree events and its complement.



(2nd Pattern) (Complement of 2nd Pattern)

Fig. 4. Phase Automata for the second pattern of fault tree events and its complement.

Complement construction for Sequence Pattern. In general, Phase Automata are not closed under complementation and there are examples of Phase Automata for which no complement exists and which belong to the group of sequence formulae. To avoid these problems, we require that two state expressions within the pattern cannot be satisfied at the same time, that is

$$\pi_i \wedge \pi_j \equiv false \quad \text{for all } i \neq j. \tag{1}$$

A construction for this case is sufficient for our case studies. The detailed construction is given in the appendix. The restriction to this kind of formula unfortunately disallows formulae like

$$\Diamond_L \lceil\neg\pi\rceil; \lceil\pi\rceil \wedge \ell < t; \lceil\neg\pi\rceil$$

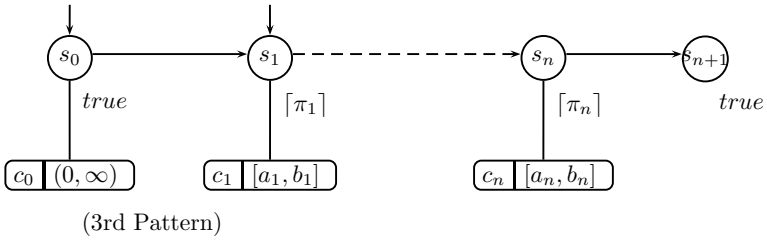


Fig. 5. Phase Automaton for the third pattern of fault tree events.

which describes that eventually π holds for less than t time units. This is the only way to describe upper bounds in DC. But this simple type of formula can still be translated although requirement (1) does not hold.

5.2 Decomposition Gates

Using the Phase Automata constructed in the previous subsection, we can express the *negation* of the different proof obligations by a set of Phase Automata as presented in table 1. Then we check if this set of Phase Automata has a run together with the Phase Automata model $\mathcal{M}$ of our system. If this is not the case, the proof obligation is verified. In table 1, $\overline{\mathcal{A}}$ denotes the complement of the Phase Automaton $\mathcal{A}$, $\|$ denotes the parallel composition, and $\vee$ the alternative. The alternative $\mathcal{A}_1 \vee \mathcal{A}_2$ is –like for finite automata– just the union of the two Phase Automata, where the name-spaces are disjoint.

Condition	Set of Phase Automata (Negation of Condition)
$F_S^{\wedge} = \left(\bigwedge_{i=1}^{n} E_i \right) \Rightarrow E$	$\overline{\mathcal{A}}_E \| \mathcal{A}_{E_1} \| \ldots \| \mathcal{A}_{E_m}$
$F_N^{\wedge} = E \Rightarrow \left(\bigwedge_{i=1}^{n} E_i \right)$	$\mathcal{A}_E \| (\overline{\mathcal{A}}_{E_1} \vee \ldots \vee \overline{\mathcal{A}}_{E_m})$
$F_S^{\vee} = \left(\bigvee_{i=1}^{n} E_i \right) \Rightarrow E$	$\overline{\mathcal{A}}_E \| (\mathcal{A}_{E_1} \vee \ldots \vee \mathcal{A}_{E_m})$
$F_N^{\vee} = E \Rightarrow \left(\bigvee_{i=1}^{n} E_i \right)$	$\mathcal{A}_E \| \overline{\mathcal{A}}_{E_1} \| \ldots \| \overline{\mathcal{A}}_{E_m}$

Table 1. Model-checking gate conditions

5.3 Cause-Consequence Gates

To perform model-checking for cause-consequence gates, we use the same automata constructed above, but impose a new oberservable *Syn* to synchronise the automata and express the temporal dependencies. Obviously the second pattern expressing a final state cannot be regarded as a cause for an event which takes place *after* this event, so we do not consider this type of pattern for cause-consequence gates.

Asynchronous Gates. The sufficient condition is the same for asynchronous cause-consequence gates and for the decomposition gates.

To check the necessary condition we construct a set of Phase Automata which allow all runs which violate the property. For the and-gate condition $F_N^{\wedge\text{-acc}} \stackrel{df}{=} \neg(\neg \bigwedge_{i=1}^{n} \Diamond E_i\{\Diamond_L/\}; E\{\Diamond_L/\})$ the construction is as follows:

1. Let Syn be a fresh boolean observable.
2. Construct the union of $\overline{\mathcal{A}_{E_1}}, \ldots, \overline{\mathcal{A}_{E_n}}$. The semantics of this automaton is the set of all runs in which at least one formula E_i is not *true*. Add the assertion Syn to every state. Add a new state p_{true} with the assertion $\neg Syn$ and transitions from all other states to this state. So Syn is *true* as long as at least one formula E_i is not satisfied.
3. Construct the automaton $\mathcal{A}_E$ and replace the condition *true* in the first state by Syn and add the assertion $\neg Syn$ to all other states. So E has to be *true* finally and Syn must hold before, so at least one formula E_i has not been true before.

The construction for or gates is similar, except that in step 2 instead of the union the parallel composition is used.

Synchronous Gates. Again, checking the sufficient condition is very easy and is skipped here. The necessary condition for synchronous and gate describes that an event occurs only if the sub-events occur at the same time, which means that several state assertions hold in the same interval. Therefore we only consider cases where the formulae E_i for the sub-events are of the first type $\Diamond_L \lceil \pi^{(i)} \rceil \wedge a^{(i)} \leq \ell$.

1. Let Syn be a fresh boolean observable.
2. Construct an automaton for the complement of the formula $\Diamond_L(\lceil \bigwedge_{i=1}^{n}(\pi^{(i)}) \rceil \wedge \bigwedge_{i=1}^{n}(a^{(i)} \leq \ell))$ using the construction in figure 3. The semantics of this automaton is the set of all runs in which not all state assertions are true simultaneously. Add the assertion Syn to every state. Add a new state p_{true} with the assertion $\neg Syn$ and transitions from all other states to this state. So Syn is *true* as long as not all state assertions are true on the same time interval.
3. Construct the automaton $\mathcal{A}_E$ and replace the condition *true* in the first state by Syn and add the assertion $\neg Syn$ to all other states. So E has to be *true* finally and Syn must hold before, which means that beforehand at no time interval all state assertions in $E_1, \ldots, E_n$ have been *true*.

6 Example – Pressure Tank

We apply our approach to the classical pressure tank example [VGRH81]. In the original work, the fault tree analysis is done completely manually; no formal techniques are considered. We present the scenario, a part of our formal model of the system, the first part of the fault tree, and explicitly check one gate condition.

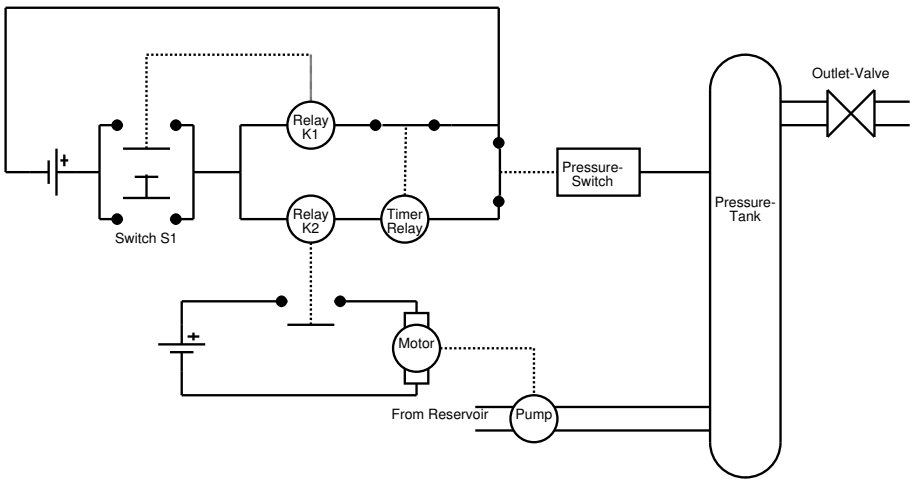


Fig. 6. Pressure Tank System

6.1 Scenario

The pressure tank system shown in figure 6 consists of three parts: a pressure tank, a pump-motor device and an associated control system to regulate the operation of the pump. We use the following assumptions of the system [VGRH81]:

- It takes 60 sec to pressurise the tank.
- The pressure switch contacts are closed until the threshold pressure is reached.
- The tank is fitted with an outlet valve which drains the entire tank in negligible time. The valve is not a pressure relief valve.

Then the operation of this system is as follows:

- Initially the system is dormant: the switch S1 contacts are open, the relay K1 contacts are open, the relay K2 contacts are open and the pressure switch is closed.
- Pressing switch S1 starts the system. Power is applied to the coil of relay K1, closing K1's contacts, so that relay K1 is electrically self-latched.
- The closure of relay K1 applies power to the coils of relay K2, causes relay K2's contacts to close and starts the pump.
- When the threshold pressure is reached, the pressure switch contacts open, deenergise the coil of relay K2, cause relay K2's contacts to open and stop the pump.
- The timer relay serves as a fall-back device if the pressure switch fails to open. After 60 sec of continuous power application to the timer relay, the contacts open, deenergise the coils of relay K1 and causes its contacts to open. Then the coil of relay K2 is deenergised, its contacts fall open and the pump is stopped.

6.2 Modelling

We use the following observables to model the state of the pressure tank system:

- *tankstate* ranging over the set $\{empty, fill, full, rupture\}$ to model the filling state of the pressure tank.
- *flowstate* ranging over $\{flow, noflow\}$ to model whether the fluid is pumped into the tank or not.
- $K2Coil$ ranging over $\{K2EMF, K2noEMF\}$ to model whether there is an electromagnetic field on the coil.
- $K2Contacts$ ranging over $\{K2open, K2closed\}$ to model whether the contacts are open or closed.

For simplicity, additional oberservables for the states of the other components are skipped here. Phase Automata are used to model our assumptions on the behaviour of the system. The ones presented in figure 7 to figure 9 model the operation of relay K2, the tank and the assumption that the tank will not withstand more than 60 sec of continuous flow. The possible failure of relay K2 is modelled by an extra failure state. The rest of our system model is again skipped.

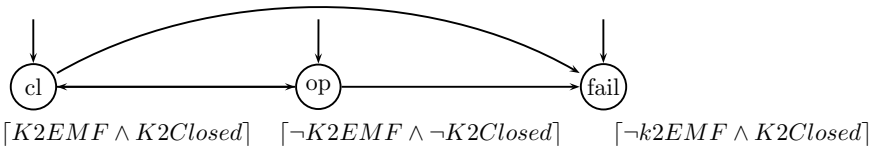


Fig. 7. Phase Automaton $\mathcal{A}_{\mathsf{K2}}$ modelling Relay K2

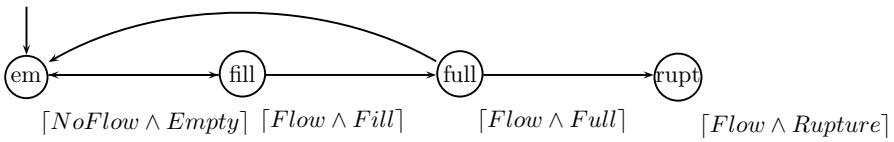


Fig. 8. Phase Automaton $\mathcal{A}_{\mathsf{tank}}$ modelling the behaviour of the tank.

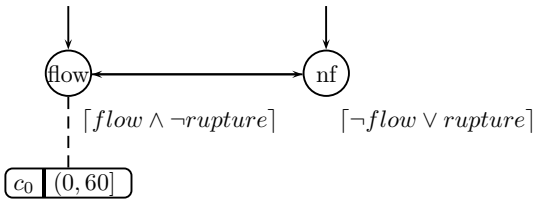


Fig. 9. Phase Automaton $\mathcal{A}_{\mathsf{ruptTime}}$ to model certain rupture after 60 sec of continuous flow.

6.3 Fault Tree Analysis

Figure 10 presents a simplified and shortened version of the fault tree developed by Veseley et al. [VGRH81]. Additionally, we have annotated every event with its DCL formula.

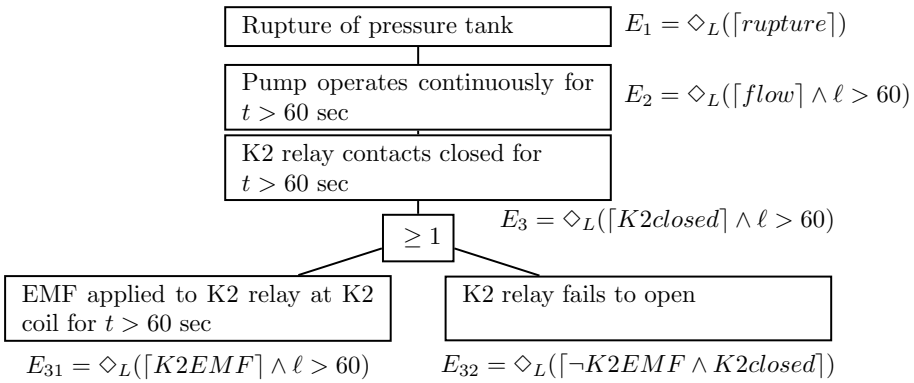


Fig. 10. First part of fault tree for pressure tank system and the event "Rupture of pressure tank"

6.4 Verification

We are going to verify that the decomposition at the or gate is correct with respect to our model of the system. That means that the Events E_{31} and E_{32} are necessary for event E_3. To this end, we have to check the validity of

$$\mathcal{M} \Rightarrow (E_3 \Rightarrow (E_{31} \vee E_{32}))$$

where $\mathcal{M}$ is our model of the system in terms of Phase Automata. Therefore we use the construction given in section 5 for the first pattern to obtain Phase Automata $\mathcal{A}_{E3}, \mathcal{A}_{\neg E31}, \mathcal{A}_{\neg E32}$ representing E_3, $\neg E_{31}$ and $\neg E_{32}$ as given in figure 11 and check whether they have a common run together with the automata of our system model. In fact we only need $\mathcal{A}_{\mathsf{K2}}$ of our model to prove this. The answer is obtained in 1.2 seconds using the tool Moby/DC.

This result holds only because we have neglected the time the relay K2 takes to open its contacts. If we considered this in our model, the implication would not hold any longer. Using this technique the engineer has to put all her assumptions on the behaviour of the system in the formal model which adds additional safety as implicit assumptions are discovered. On the other hand, the engineer can easily alter the model and check whether the fault tree remains correct under different assumptions.

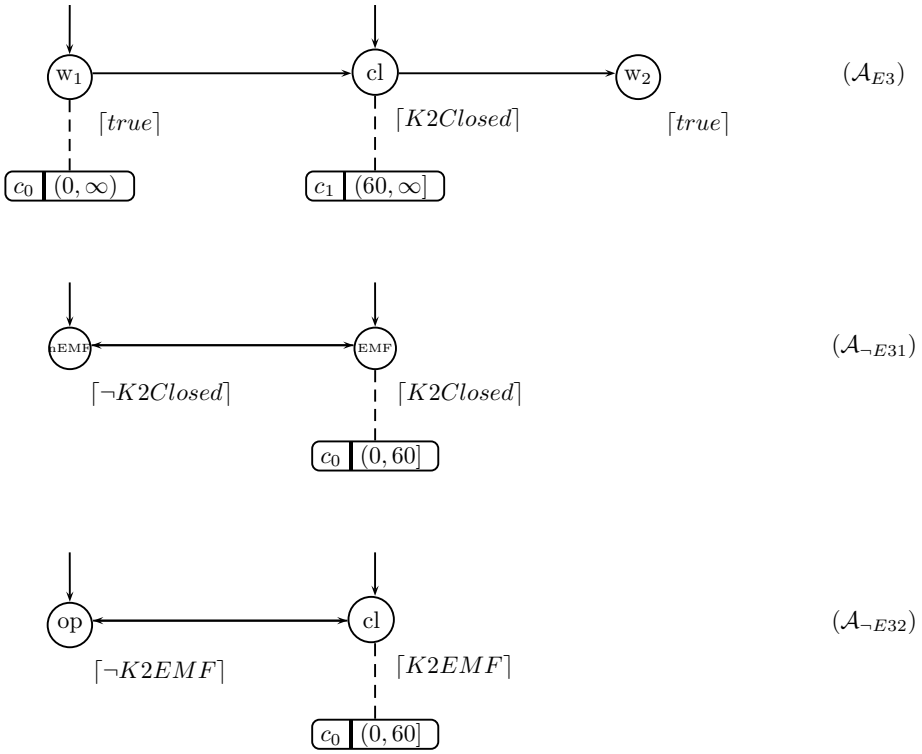


Fig. 11. Phase Automata corresponding to the events $E_3, \neg E_{31}$ and $\neg E_{32}$

7 Combination with other Model-Checking Techniques

In the previous section we have shown how an engineer can benefit from the combination of fault tree analysis and real-time model-checking. In this section we look at the profit gained from the model-checking point of view. We demonstrate how fault tree analysis can be used as a decomposition method to allow model-checking of larger systems. The case study is the single track line segment [STL01].

7.1 Scenario

Two trains drive on the tracks shown in figure 12. On the outermost track the trains may go clockwise and on the innermost track counterclockwise. In the critical section trains may go in both directions and may change their direction once. The task is to design a distributed controller ensuring that no collision may happen in the critical section. Each component of the controller has three sensors (S) attached and controls one light signal (L) and one point. This controller has to allow two trains to pass the critical section one directly after the other. In this case the first train may not change its direction.

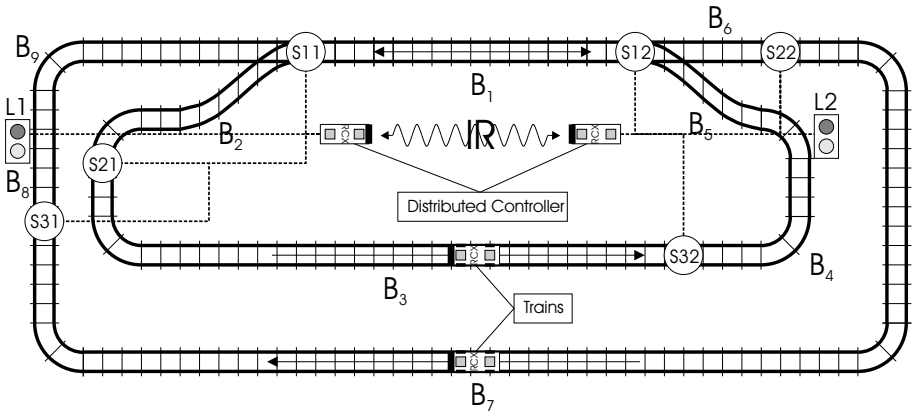


Fig. 12. Single Track Line Segment Scenario

7.2 Design

We built a real-life model of this case study using the Lego-Mindstorms and the open source operating system BrickOS We designed the controller using PLC-Automata [Die00], which also have a semantics in DC. Using the tool Moby/PLC [TD98] these automata can be compiled into ST-Code for Programmable Logic Controller, into C++ Code for BrickOS (Lego-Mindstorms), and into Timed Automata [AD94]. We used the compilation into C++ Code for BrickOS.

7.3 Verification

The goal is to verify that two trains do not collide in the critical section. The obvious idea would be to compile the PLC-Automata for the distributed controller into Timed Automata. Then one would model the environment using Timed Automata, and finally use the model-checker Uppaal [BBD+02] to verify that a collision in the critical section is impossible. But the model is too complex and hence direct model-checking failed. So instead we choose the following approach which is sketched in figure 13.

We perform a fault tree analysis with the top event "collision of two trains in critical section". In the fault tree this top-event is iteratively decomposed until we obtain a number of basic events. For each gate in the fault tree we apply the technique described in section 5, i.e. we translate the events into Phase Automata and verify, using Moby/DC, that for each decomposition the sub-events are necessary for the upper event. The fault tree for this example consists of 38 events and 27 gates. It turns out that due to symmetry only 14 gate conditions have to be checked.

For each basic event we verify that it cannot occur. First, all basic events in which the first of two subsequent trains turns around in the critical section may not occur, simply because this behaviour is forbidden by the specification.

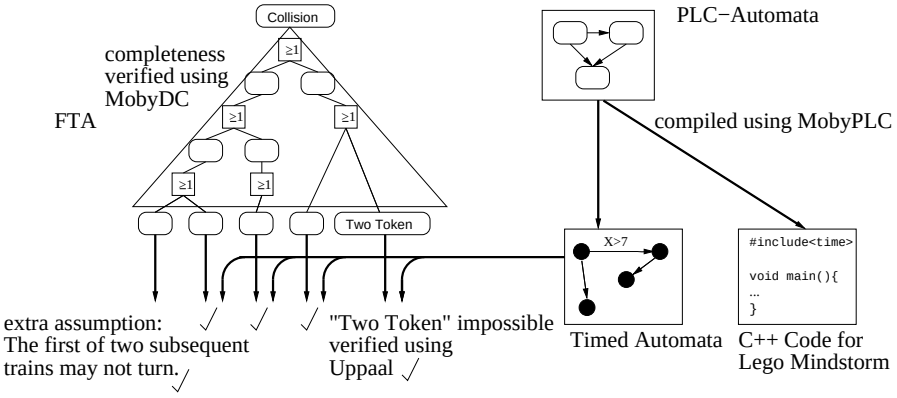


Fig. 13. Verification approach

Second, all other basic events are simple enough for automatic verification. We show that they cannot occur in the distributed controller modelled by PLC-Automata. To this end, we use MobyPLC to compile these automata into Timed Automata which are then checked by Uppaal against the basic events. Since none of the events is possible in the controller model, we conclude that the top-event, i.e. the collision, does not occur.

Modelling. Our formal model of the single track line segment system in terms of Phase Automata describes the topology of the tracks and the movement of the two trains.

Experimental Results. The verification that a basic event cannot occur took 1:04:37 h for the hardest one. We used Uppaal (Version 3.2.11) on a Dual-Pentium with 450 Mhz and 1 GB RAM. Checking each gate condition takes about 10 seconds on a Sun Ultra-1 with 384 MB RAM using Phase Automata and Moby/DC.

8 Related Work

There are several approaches to define formal semantics for fault tree analysis. Special timed transition systems and a first order logic with special predicates are introduced by Gorski [Gór94]. Dugan et al. [DBB93] introduced Markov Models to resolve ambiguities. Bruns and Anderson [BA93] use a modal μ-calculus semantics to check the validity of formal system models.

Hansen [Han96] gives a Duration Calculus semantics and uses fault tree analysis to derive safety requirements from a given fault tree. However, the work does not consider whether a fault tree is constructed properly.

In the FORMOSA project [RST00,STR02] semantics in Duration Calculus, CTL and ITL are considered. Discrete time model-checking, using Raven [Ruf01]

and SMV, and fault tree analysis have been applied to several case studies but they are used rather independently and not tightly integrated; further integration is one aim of this project. Currently embedding fault tree analysis in the interactive theorem prover KIV is faced.

The ESACS project (http://www.cert.fr/esacs/) uses fault tree analysis and model-checking in different areas. It is used for test-case generation from fault trees and for compilation of mode automata into a boolean formula, which is presented as a fault tree. Furthermore a tool for the automatic generation of fault trees from a statemate model is developed. But neither order of events nor time is considered in current versions of this tool.

9 Conclusion and Future Work

We have shown how fault tree analysis can be turned into a formal method and how model-checking can be applied to prove necessary and sufficient conditions of this analysis. In the case study we integrated fault tree analysis with two other formal techniques, PLC-Automata and Timed Automata, to verify a larger system.

In our future work we would like to investigate whether we captured all usual cases of events which might occur in fault trees. We also would like to implement tool support. This tool should compile a given fault tree into Phase Automata and check which gate conditions hold and which do not. Translation into other operational models like Timed Automata may also be considered.

Acknowledgements

Our paper is inspired by the work of W. Reif, G. Schellhorn and A. Thums of Augsburg University. The author thanks E.-R. Olderog, H. Dierks, and M. Möller for draft-reading earlier versions and many helpful remarks and the members of the group "Correct System Design" at Oldenburg University for fruitful discussions and comments.

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BA93] G. Bruns and S. Anderson. Validating safety models with fault trees. In *SAFECOMP '93: the 12th international Conference on Computer Safety*, pages 21–30. Springer, 1993.

[BBD+02] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and Wang Yi. Uppaal implementation secrets. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2002*, volume 2469 of *LNCS*, pages 3–22, 2002.

[DBB93] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. Fault trees and markov models for reliablility analysis of fault-tolerant digital systems. *Reliability Engineering and System Safety*, 39:291–37, 1993.

[Die00] H. Dierks. PLC-automata: A new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, 2000.

[DT03] H. Dierks and J. Tapken. Moby/DC – a tool for model-checking parametric real-time specifications. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems 2003*, volume 2619 of *LNCS*, pages 271–277, 2003.

[Gór94] J. Górski. Extending safety analysis techniques with formal semantics. In F. Redmill, editor, *Technology and assessment of safety-critical systems: proceedings of the Second Safety-Critical Systems Symposium*, pages 147–163. Springer Verlag Berlin, 1994.

[Han96] K. M. Hansen. *Linking Safety Analysis to Safety Requirements*. PhD thesis, Institut for Informationsteknologi, DTU Lyngby, 1996.

[IEC93] IEC 61025: Fault tree analysis, 1993.

[RST00] W. Reif, G. Schellhorn, and A. Thums. Safety analysis of a radio-based crossing control system using formal methods. In *Proceedings of the 9th IFAC Symposium Control in Transportation Systems 2000, June 13-15, Braunschweig, Germany*, 2000.

[Ruf01] J. Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *Journal of Universal Computer Science*, 7(1):89–104, January 2001.

[Sch02] A. Schäfer. Fault tree analysis and real-time model-checking. Master's thesis, University of Oldenburg, 2002. in German.

[Ska94] J. U. Skakkebæk. Liveness and fairness in duration calculus. In B. Jonsson and J. Parrow, editors, *CONCUR'94*, volume 836 of LNCS, pages 283–298. Springer-Verlag, 1994.

[STL01] Practical course real-time systems: Final report. `http://csd.informatik.uni-oldenburg.de/teaching/fp_realzeitsys_ws0001/result/eindex.html`, 2001.

[STR02] Gerhard Schellhorn, Andreas Thums, and Wolfgang Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology, Pasadena, CA,*, 2002.

[Tap01] J. Tapken. *Model-Checking of Duration Calculus Specifikations*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2001.

[TD98] J. Tapken and H. Dierks. Moby/PLC – graphical development of PLC-automata. In A.P. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems 1998*, volume 1486 of *LNCS*, pages 311–314. Springer Verlag, 1998.

[VGRH81] W.E. Veseley, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook*. Washington DC: US Nuclear Regulatory Commission, NUREG-0492, 1981.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZHS93] Zhou Chaochen, M. R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *STACS 93, 10th Annual Symposium on Theoretical Aspects of Computer*, volume 665 of *LNCS*, pages 58–68, 1993.

A Complement-Construction for Sequence-Pattern

We give a construction for the complement of an automaton corresponding to the sequence pattern $\Diamond_L(\lceil\pi_1\rceil \wedge a_1 \sim \ell; \lceil\pi_2\rceil \wedge a_2 \sim \ell \sim b_2; \ldots; \lceil\pi_{n_1}\rceil \wedge a_{n-1} \sim \ell \sim b_{n-1}; \lceil\pi_n\rceil \wedge a_n \sim \ell)$ and $\pi_i \wedge \pi_j \equiv false$ for all $i \neq j$. The cases where the relation $<$ occurs are analogous. This case is simpler than the more general one where only $\pi_i \wedge \pi_{i+1} \equiv false$ is required. But the sequences which occured in the case study presented in this paper were of this simpler type. For each state assertion π_i which occurs in the given sequence we create four states.

- p_i which is taken iff the assertion π_i holds and the sequence up to π_i has not yet been seen.
- p_i^* which is taken iff the assertion π_i holds and the sequence up to π_i has been seen.
- $p_{i<}$ iff π_i holds and the duration is too short.
- $p_{i>}$ iff π_i holds and the duration is too long.

Additionally we have a state p_{else} which is taken iff no state assertion in $\pi_1, \ldots, \pi_n$ holds. Let $\overline{\mathcal{A}_S} = (P, E, C, cl, s, d, P_0)$. The state space and transition relation is defined by

$$P \stackrel{df}{=} \{p_2, \ldots, p_n, p_1^*, \ldots, p_{n-1}^*, p_{1<}, \ldots, p_{n<}, p_{2>}, \ldots, p_{(n-1)>}, p_{else}\}$$

$$\begin{aligned}
E \stackrel{df}{=} & \{p_i \to p_j | i \neq j\} \cup \{p_i \to p_{j<} | i \neq j\} \cup \{p_i \to p_{j>} | i \neq j\} \cup \{p_i \to p_{else}\} \\
& \cup \{p_i \to p_1^*\} \\
& \cup \{p_{i<} \to p_j | i \neq j\} \cup \{p_{i<} \to p_{j<} | i \neq j\} \cup \{p_{i<} \to p_{j>} | i \neq j\} \\
& \cup \{p_{i<} \to p_{else}\} \cup \{p_{i<} \to p_1^*\} \\
& \cup \{p_{i>} \to p_j | i \neq j\} \cup \{p_{i>} \to p_{j<} | i \neq j\} \cup \{p_{i>} \to p_{j>} | i \neq j\} \\
& \cup \{p_{i>} \to p_{else}\} \cup \{p_{i>} \to p_1^*\} \\
& \cup \{p_i^* \to p_{i+1}^* | (i+1) < n\} \cup \{p_i^* \to p_j | i \neq j \wedge i+1 \neq j\} \cup \{p_i^* \to p_{j<} | i \neq j\} \\
& \cup \{p_i^* \to p_{j>} | i \neq j\} \cup \{p_i^* \to p_1^*\} \cup \{p_i^* \to p_{else}\} \\
& \cup \{p_{else} \to p_i\} \cup \{p_{else} \to p_{i<}\} \cup \{p_{else} \to p_{i>}\} \cup \{p_{else} \to p_1^*\}
\end{aligned}$$

We associate exactly one clock to each state. The state assertions for each state and the initial states and the assigned clock intervals are defined as follows.

$$s(p) \stackrel{df}{=} \begin{cases} \pi_i & \text{if } p = p_i \\ \pi_i & \text{if } p = p_i^* \\ \pi_i & \text{if } p = p_{i<} \\ \pi_i & \text{if } p = p_{i>} \\ \neg \bigvee_{i=1}^n \pi_i & \text{if } p = p_{else} \end{cases} \qquad d(p) \stackrel{df}{=} \begin{cases} [b_i, e_i] & \text{if } p = p_i \\ [b_i, e_i] & \text{if } p = p_i^* \\ (0, b_i) & \text{if } p = p_{i<} \\ (e_1, \infty] & \text{if } p = p_{i>} \end{cases}$$

$$P_0 \stackrel{df}{=} \{p_2, \ldots, p_n, p_1^*, p_{1<}, \ldots, p_{n<}, p_{2>}, \ldots, p_{(n-1)>}, p_{else}\}$$