

Model Checking FTA

A. Thums and G. Schellhorn

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{thums,schellhorn}@informatik.uni-augsburg.de

Abstract. Safety is increasingly important for software based, critical systems. Fault tree analysis (FTA) is a safety technique from engineering, developed for analyzing and assessing system safety by uncovering safety flaws and weaknesses of the system. The main drawback of this analysis technique is, that it is based on informal grounds, so safety flaws may be overlooked. This is an issue, where formal proofs can help. They are a safety techniques from software engineering, which are based on precise system descriptions and allow to prove consistency and other (safety) properties.

We present an approach which automatically proves the consistency of fault trees based on a formal model by model checking. Therefore, we define consistency conditions in Computational Tree Logic, a widely used input language for model checkers. In the second part, we exemplify our approach with a case study from the *Fault Tree Handbook*.

Keywords: model checking, safety analysis, fault tree analysis

1 Introduction

Safety is an important issue for software based, critical systems like aerospace, nuclear power plants, transportation, and medical control. In engineering, fault tree analysis (FTA, [23]) is commonly used to analyze system safety. It breaks down system level hazards to failures of components, called (failure) events. The events are connected through gates, indicating if all or only any sub-event is necessary to cause the failure. Each event is analyzed recursively, resulting in a tree of events. The leafs of a fault tree describe failures of basic components which in combination cause the system hazard.

FTA is based on an informal description of the underlying system. Therefore it is quite hard to check the consistency of the analysis. It is possible that causes are noted in the tree which do not lead to the hazard (in-correctness) and, more critical, that some causes for the hazard are overlooked during analysis (in-completeness).

This is an issue, where formal proofs can help. They are a formal method used in software engineering (although there are applications in hardware design, too) based on a specification of the system behavior in a precise and unambiguous

notation. The specification is validated through proving required system properties. Verification guarantees safety properties and functional correctness. We will use model checking to automate these proofs.

The idea of the integration of FTA and formal methods is to formally specify the system model and to prove correctness and completeness of FTA. Therefore, the FTA has to be formalized. We define a semantics of fault trees which consists of proof conditions for every gate in the tree. We assign so called *correctness* and *completeness* conditions to the gates. The correctness condition guarantees, that the noted sub-events *actually* lead to the top event and the completeness condition, that *only* the sub-events lead top event, i.e. no cause was overlooked during the FTA. If every proof condition is verified, the correctness and completeness of the fault tree is guaranteed, i.e. the basic events are necessary for causing the system level hazard and only the basic events can cause the hazard. We call this integrated approach ‘formal FTA’. For applying formal FTA we have to i) formally specify the system, ii) develop fault trees for the main system hazards, iii) formalize the fault tree events, and iv) verify the proof conditions for every gate.

Step iii) will formalize events by temporal logic formulae. Note, that first order formulae are not sufficient, since we do not use the term ‘event’ as it is used in automata theory, where an event happens at a fixed point in time and has no duration. Events in safety engineering may also describe a system state or sequence of states and can last some time¹. E.g. ‘10 sec flow of fluid’ can describe a failure event. In the following, we use ‘event’ with the more general meaning and distinguish between events with duration and events without duration by the terms *temporal* and *timeless* events, respectively.

We presented the approach of formal FTA in [20] and defined an ITL semantics for general system descriptions. But in industry, formal methods are accepted best, if verification can be automated. Model checkers can do this for finite state systems and additionally help to find failures by generating counterexamples. When validating fault trees, in most cases these counterexamples point to overlooked causes for the system hazard. Therefore, we develop a FTA semantics in Computational Tree Logic (CTL) in this paper. CTL is widely used as an input language for model checkers, e.g. in SMV [10], SVE [5], and RAVEN [14]. The drawback of CTL is, that its tree structure does not allow to treat temporal events directly in the logic. We introduce so called ‘observer automata’ to cope with temporal events. They extend the system model and enter an acceptance state, if they observed the corresponding temporal event. Entering the acceptance state is a timeless event and can be treated correctly.

To exemplify our approach, we prove the correctness and completeness conditions for an example of the *Fault Tree Handbook* [23], the ‘pressure tank’. This example extensively uses time intervals for its specification, which are supported by the model checker RAVEN. RAVEN uses timed automata [15, 18] for specifying the system model and an extended CTL, clocked CTL (CCTL, [17]), to

¹ As noted by Leveson [9], this is a common misunderstanding between safety engineers and (theoretical) computer scientists.

specify the proof conditions. Using timed automata makes the specification of the example simpler and better understandable.

The paper is organized as follows. The ‘pressure tank’ example is introduced in Sect. 2. In Sect. 3 we describe the informal FTA and present the fault tree of the pressure tank example from [23]. After this informal analysis, we consider the formal aspects. Logical foundations for formalizing FTA and specifying the pressure tank example are given in Sect. 4. We shortly introduce CTL, CCTL, and timed automata. Section 5 describes the formal CTL semantics of FTA. To cope with temporal events we introduce observer automata, which maps temporal to timeless events. After this theoretical part, we present the formal FTA of the pressure tank example in Sect. 6. Section 7 discusses differences and communities with other formalizations of FTA from literature and Sect. 8 concludes the paper.

2 Example: Pressure Tank

The pressure tank is an example from the *Fault Tree Handbook* [23, Chapter VIII]. It describes a control system regulating the operation of a pump, which fills fluid into a tank.

The control system is depicted in Fig. 1. It consists of a control part, powered by P_1 , and an electrically isolated electrical circuit, powered by P_2 , which energizes the motor of the pump. P_1 powers the electrical circuits c_1 and c_2 . c_1 energizes the coil of relay K_1 , if the push button S_1 or K_1 and the timer T are closed. The circuit c_2 energizes the coil of relay K_2 and the timer and is closed, if S_1 or K_1 and the pressure switch S are closed. P_2 powers c_3 which supplies power to the motor, if the relay K_2 is closed.

The control system starts operation when the push button S_1 is pressed. This closes the electrical circuit c_1 and applies power to relay K_1 , which henceforth energizes the control system.

If the pressure tank is empty and, therefore, the pressure switch S is closed, the electrical circuit c_2 applies power to relay K_2 , energizing electrical circuit c_3 and starting the motor of the pump. This situation is sketched in Fig. 1. We assume, that it takes 60 sec to pressurize the tank. When the tank is full, the pressure switch S is opening, de-energizing electrical circuit c_2 , which in effect opens relay K_2 and electrical circuit c_3 . The pump motor is switched off and the pump stops filling. Now the outlet valve may drain the tank and, if the tank is empty, the cycle described above is repeated. So, in normal operation without failures, only the pressure switch S and the relay K_2 are opening and closing, depending on the liquid level of the tank.

If the pressure switch is defective, the timer is a fall-back to prevent tank rupture. If the pressure switch fails to open, although the tank is full, the timer sends a timeout. The timer registers, if the circuit c_2 is continuously energized for more than 60 sec and provides an emergency shutdown by de-energizing electrical circuit c_1 . The timer T opens and de-energizes K_1 . Then, neither S_1 nor K_1 are closed and the system has to be restarted by pressing S_1 . In normal

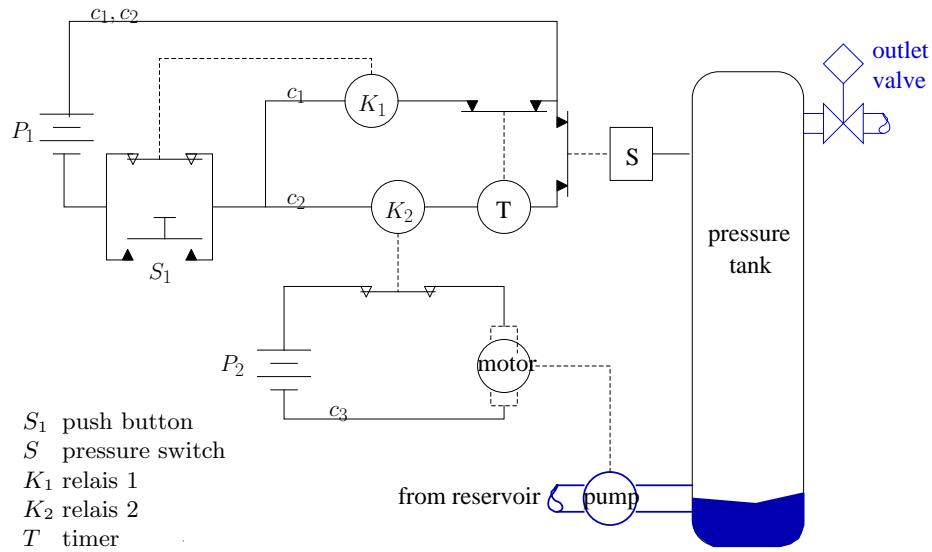


Fig. 1. Pressure Tank

operation, when the pressure switch S de-energizes the electrical circuit c_2 , the timer is reset.

3 Fault Tree Analysis (FTA)

Fault tree analysis (FTA) is a well known technique in engineering and was developed for technical systems to analyze, if they permit a hazard (top event). This event is noted at the root of the fault tree. Events, which cause the hazard (intermediate events), are given in the child nodes and analyzed recursively, resulting in a tree of events. Each analyzed event (main event, either top or intermediate) is connected to its causes (sub-events) by a gate in the fault tree (see Fig. 2). An AND-gate indicates that all sub-events are necessary to trigger the main event, for an OR-gate only one sub-event is necessary. An INHIBIT-gate states that in addition to the cause stated in the sub-event the condition (noted in the oval) has to be true to trigger the main event. The INHIBIT-gate is more or less an AND-gate with one cause and a condition, but the condition needs not to be a fault event.

The leaves of the tree are the low level causes (basic events) for the top event, which have to occur in combination (corresponding to the gates in the tree) to trigger the top event. A set of basic events together leading to the hazard is called a *cut set*. A cut set is minimal, if it has no other cut set as a subset. I.e. a *minimal cut set* can not lead to the top level hazard, if at least one event of the set is prevented. Minimal cut sets can be computed from fault trees by combining

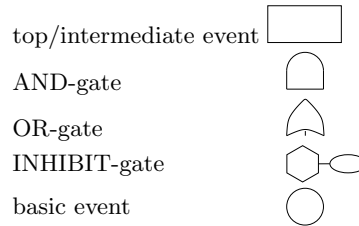


Fig. 2. Fault Tree Symbols

the basic events with boolean operators as indicated by the gates. A minimal cut set then consists of the elements of one conjunction of the disjunctive normal form of the resulting formula.

Cut sets are used to identify failure events, which have a big impact on system safety. E.g., if one event occurs in different minimal cut sets, the probability of the top level hazard will strongly decrease, if this event can be excluded. Besides this *qualitative* analysis, FTA also allows *quantitative* analysis. If the failure probabilities of the basic events are known (failure probabilities of basic components can frequently be derived from statistical data) and the events occur only once in the fault tree, the probability of the occurrence of the hazard can be computed starting from the cut sets. For statistically independent failures, the failure probability of the main event is the sum of the probabilities of the minimal cut sets (for small values). These probabilities are the product of the failure probabilities of its events. For more sophisticated computations, we refer to the Fault Tree Handbook [23]. Therefore, FTA is a good starting point for the assessment of system safety.

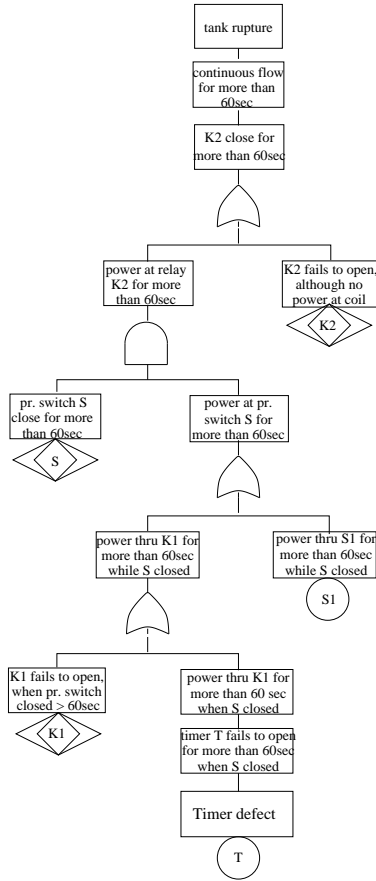
Figure 3 describes the fault tree for the pressure tank example (see [23], page VIII-11). It analyses the hazard ‘rupture of pressure tank after the start of pumping’. *Rupture* occurs, if the tank is *continuously filled for more than 60 sec*. The reason therefore is, that relay K_2 is *closed for more than 60 sec*. This could be the effect of *power at the coil of K_2 for more than 60 sec* or of a defective relay K_2 (fails to open). Both causes are connected through an OR-gate to the effect. The rest of the fault tree is developed in the same manner and the resulting cut sets are

$$\{\{K_2, \}, \{S, S_1\}, \{S, K_1\}, \{S, T\}\}.$$

The qualitative analysis of the cut sets shows, that relay K_2 is a single point of failure. With the failure probabilities for the basic events from [23] (see Fig. 3 on the right), we can quantify the system safety and compute the overall hazard probability to approximately $3 * 10^{-5}$.

4 Logical Foundations

We will formalize the semantics of FTA in CTL, because many model checkers use CTL as an input language for the properties to prove. We ourselves used



failure probabilities:

components:

$P(K_2)$	$3 * 10^{-5}$
$P(S)$	$1 * 10^{-4}$
$P(K_1)$	$3 * 10^{-5}$
$P(T)$	$1 * 10^{-4}$
$P(S_1)$	$3 * 10^{-5}$

cut sets:

K_2	$3 * 10^{-5}$
$\{S, S_1\}$	$3 * 10^{-9}$
$\{S, K_1\}$	$3 * 10^{-9}$
$\{S, T\}$	$1 * 10^{-8}$
total	$\approx 3 * 10^{-5}$

Fig. 3. FTA: Pressure Tank

the model checker RAVEN [14] for checking correctness and completeness of the pressure tank example. RAVEN uses timed automata for system specification and an extension of CTL, so called Clocked CTL (CCTL) for specifying the proof conditions. In the following we shortly introduce CTL and subsequently the extensions of RAVEN, CCTL and timed automata.

4.1 CTL

The semantics of CTL [4] is based on state transition diagrams. A state transition diagram is a tuple $(\mathcal{P}, \mathcal{S}, \mathcal{T}, \mathcal{L})$ with \mathcal{P} a set of atomic formulas, \mathcal{S} a finites set of states, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ a finite set of transitions, and $\mathcal{L} : \mathcal{S} \rightarrow \wp(\mathcal{P})$ a label function. A

state s is labeled with atomic formulas $\varphi \in \mathcal{P}$ and a formula holds in s ($s \models \varphi$), if and only if $\varphi \in \mathcal{L}(s)$.

Possible successors s' of a state s are described in the transition relation \mathcal{T} , i.e. $(s, s') \in \mathcal{T}$ and we require, that every state s has a successor s' with $(s, s') \in \mathcal{T}$. A path is an infinite sequence of states $\sigma = (s_0, s_1, \dots)$ with $\sigma[i] = s_i$ and $(s_i, s_{i+1}) \in \mathcal{T}$ for every $i \in \mathbb{N}$. $\Pi(s)$ is the set of all possible paths starting in state s . CTL defines the following operators:

Definition 1 *CTL operators*

$$\begin{aligned} s \models EX\varphi & \quad :\Leftrightarrow \text{there exists a state } s' \in \mathcal{S} \\ & \quad \text{with } (s, s') \in \mathcal{T} \text{ and } s' \models \varphi \\ s \models EG\varphi & \quad :\Leftrightarrow \text{there exists a path } \sigma \in \Pi(s), \\ & \quad \text{s.t. for every } k \in \mathbb{N}: \sigma[k] \models \varphi \\ s \models E(\varphi U \psi) & \quad :\Leftrightarrow \text{there exists a path } \sigma \in \Pi(s), \\ & \quad \text{a } k \in \mathbb{N}: \sigma[k] \models \psi, \text{ and for every } i < k: \sigma[i] \models \varphi \end{aligned}$$

Additional temporal operators can be derived:

Definition 2 *derived CTL operators*

$$\begin{aligned} EF\varphi & = E(\text{true } U \varphi) \\ E(\varphi \text{ wB } \psi) & = \neg A(\neg\varphi U \psi) \\ AX\varphi & = \neg EX\neg\varphi \\ AG\varphi & = \neg E(\text{true } U \neg\varphi) \\ A(\varphi U \psi) & = \neg E(\neg\varphi U (\neg\varphi \wedge \neg\psi)) \wedge \neg EG\neg\psi \\ AF\psi & = \neg EG\neg\psi \end{aligned}$$

One can distinguish between path operators E (there exists a path) and A (on every paths) and state operators X (next), F (eventually), G (generally), and U (until). In CTL, every state operator must be preceded by a path operator and every path operator must be followed by a state operator.

For the definition of the FTA semantics we define the precedes operator P .

Definition 3 *precedes*

$$\begin{aligned} s \models A(\varphi P \psi) & \Leftrightarrow \text{for every path } \sigma \in \Pi(s): \\ & \quad \text{for every } k \in \mathbb{N} \text{ with } \sigma[k] \models \psi, \\ & \quad \text{exists an } i \leq k \text{ with } \sigma[i] \models \varphi \end{aligned}$$

Informally, $A(\varphi P \psi)$ means for all paths, if ψ holds at some point in time, then φ must hold before or at the same point in time. If ψ never holds, φ need not hold either. The precedes operator $A(\varphi P \psi)$ is equivalent to $\neg E(\neg\varphi U (\psi \wedge \neg\varphi))$ and can therefore be specified by every CTL model checker.

4.2 Clocked CTL and timed automata

Clocked CTL (CCTL) [17] extends CTL with the possibility to annotate intervals to the state operators.

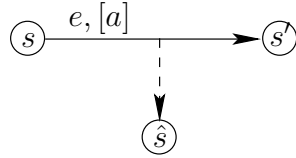
Definition 4 *CCTL state operators*

- $X_{[a]}\varphi$: φ holds after a steps
- $F_{[a,b]}\varphi$: φ holds somewhere between a and b steps
- $G_{[a,b]}\varphi$: φ holds everywhere between a and b steps
- $\varphi U_{[a,b]}\psi$: ψ occurs at t , $a \leq t \leq b$, and φ holds until t

The path quantifiers keep their usual meaning. E.g. $\varphi U_{[2,3]}\psi$ means, that on every path φ must hold until ψ and ψ is true in the second or third step. If the lower bound of an interval $[a, b]$ is zero ($a = 0$), we simply write $[b]$. For $a = 0$ and $b = \infty$ we can omit the whole interval and get the usual CTL semantics.

CCTL formulas are interpreted over timed automata [15, 18] which are state transition diagrams with the possibility to annotate input events and interval expressions. We will explain the graphical notation only.

Definition 5 *timed automata*



The input variable is e . If e holds for a time steps after entering s , state s' will be entered. If e holds, but less than a time steps, the error state \hat{s} will be entered. If e doesn't hold at all, another transition leaving state s will be taken.

If $a = 0$, the error state \hat{s} is irrelevant and we get the semantics of usual state transition diagrams.

If we want to state, that we leave state s if we can continuously watch the event e for a steps, we modify the automaton from definition 5: the dashed arrow will go back to state s , i.e. $s = \hat{s}$. By entering (the error state) s again, e 'gets a second chance' to hold a time steps. Such automata are used in the specification of the pressure tank example. General timed automata can easily be translated into untimed ones [16].

5 FTA Semantics in CTL

For model checking the validity of fault trees we develop a CTL formalization for the conditions described by the fault tree gates. Unfortunately, this formalization is adequate only, if the events in the fault tree are timeless. But Górski [6] noticed three typical patterns for temporal events in fault trees.

1. event occurs (for a certain time), e.g. '(60 sec) pumping'
2. event occurs and continues to stay, e.g. 'rupture' (can not be reversed)
3. a sequence of events (with duration) occurs with some state conditions, e.g. ' S_1 pressed, then K_1 closed, ..., and finally pumping of fluid'

All three patterns (can) describe temporal events. To handle these events (e.g. '60 sec pumping') we introduce so called observer automata. They observe an

event and accept its ending (60 sec pumping are over) by entering a corresponding state.

In the following, we present the FTA semantics in CTL and afterwards the extensions for handling temporal events. For better readability, we use CCTL and the timed automata notation for these extensions.

5.1 FTA Semantics

The aim is to develop a modular semantics for FTA. We assign proof obligations to each gate in the fault tree. They should guarantee, that if every proof obligation can be verified over the system model, the FTA is correct and complete, i.e. the causes in the cut sets actually lead to the system hazard and there exist no other causes for the system hazard. We develop proper proof obligations in the following.

The accepted informal semantics assigns the boolean semantics to each gate [23], i.e. the OR-gates are interpreted as disjunctions and the AND-gates as conjunctions. The correctness condition for a gate, which connects a cause φ with a consequence ψ , is $\varphi \rightarrow \psi$ (if the cause occurs, the consequence must occur). The cause is either $\varphi := \varphi_1 \wedge \varphi_2$ for an AND-gate or $\varphi := \varphi_1 \vee \varphi_2$ for an OR-gate. The completeness condition is $\psi \rightarrow \varphi$ (if the consequence occurs, cause must occur). In dynamic systems, these conditions have to be true for every system state, i.e. we formulate the correctness condition in CTL as $AG(\varphi \rightarrow \psi)$ and the completeness as $AG(\psi \rightarrow \varphi)$.

This boolean semantics seems adequate, but is not sufficient for dynamic systems. Consider the pressure tank example from Sect. 2. If the tank is full, but the pump continues pumping, this does not necessarily rupture the tank at once, but after a certain time. Therefore, we have to distinguish between *decomposition* gates (D-gates) with the boolean semantics and *cause consequence* gates (C-gates), where time may elapse between the causes and the consequence.

The cause consequence gates consider the case, where the causes *lead to* the consequence later on. We define correctness as $(EF \varphi) \rightarrow (EF \psi)$ and completeness as $A(\varphi P \psi)$. The correctness condition requires, that if the cause occurs eventually, the consequence has to occur as well. Unfortunately, we can formulate only a weak form of correctness which does not require that the cause φ occurs before the consequence ψ and not even that the cause occurs on the same path. But this property is guaranteed by the completeness condition. It requires, that if the consequence occurs, the cause has to occur before or at the same time. If completeness cannot be proven, there exists a run through the system, where the consequence occurs, but no mentioned cause has occurred before or at the same time. This means, there exists another cause which has been overlooked.

A distinction between decomposition and cause consequence gates is proposed in [2] as well. But we have to consider yet another case. For AND-gates we have to distinguish between so called synchronous and asynchronous AND-gates. For synchronous AND-gates, the causes have to occur together, for asynchronous AND-gates they may happen at different times. INHIBIT-gates are formalized like synchronous AND-gates. For correctness, the side-condition must happen

together with the cause. Completeness is proven for the cause only, since the condition is no fault event. Note that specifying an exact condition χ , such that $\varphi \wedge \chi$ holds before the consequence, is often difficult and not necessary for the completeness. Therefore, we require only $A(\varphi P \psi)$ and not $A(\varphi \wedge \chi P \psi)$.

Summarizing, we get 7 types of gates: D-OR-, D-AND- and, D-INHIBIT-gates (\hat{D} , \square_D , $\hat{D}-\circ$), C-OR-gates (\hat{C}), synchronous and asynchronous C-AND-gates (\square_C , \hat{C}), and C-INHIBIT-gates ($\hat{C}-\circ$). The correctness- and completeness conditions for each of these types of gates g are listed in Fig. 4. The case of two causes is shown, the generalization to any number $n \geq 1$ of causes should be obvious (for $n = 1$, AND- and OR-gates have the same meaning).

gate g	correctness CORR(g)	completeness COMPL(g)
	$AG (\varphi_1 \wedge \varphi_2 \rightarrow \psi)$	$AG (\psi \rightarrow \varphi_1 \wedge \varphi_2)$
	$AG (\varphi_1 \vee \varphi_2 \rightarrow \psi)$	$AG (\psi \rightarrow \varphi_1 \vee \varphi_2)$
	$AG (\varphi \wedge \chi \rightarrow \psi)$	$AG (\psi \rightarrow \varphi)$
	$EF (\varphi_1 \wedge \varphi_2) \rightarrow EF \psi$	$A ((\varphi_1 \wedge \varphi_2) P \psi)$
	$(EF \varphi_1 \wedge EF \varphi_2) \rightarrow EF \psi$	$A (\varphi_1 P \psi) \wedge A (\varphi_2 P \psi)$
	$EF (\varphi_1 \vee \varphi_2) \rightarrow EF \psi$	$A ((\varphi_1 \vee \varphi_2) P \psi)$
	$EF (\varphi \wedge \chi) \rightarrow EF \psi$	$A (\varphi P \psi)$

Fig. 4. semantics of fault trees

Our explanatory statements above give some indication of the adequacy of the CTL semantics. The technical report [22] details a formal proof, that for

timeless events this CTL semantics is equivalent to the more general ITL semantics presented in [20] and to other semantics from literature [7, 2]. As a corollary, the *minimal cut set theorem* proved in [20] is valid for the CTL semantics with timeless events. It guarantees, that if every event of one cut set occurs, the hazard occurs as well and if one event from every minimal cut set can be excluded, the hazard can not occur. A proven fault tree is a part of a formal safety statement (no cut set implies no hazard) for system models, which specified components failures or defects.

5.2 Events with Duration

To explain the problems of fault tree with temporal events, let us consider the topmost relation of the fault tree in Fig. 1. The cause φ is ‘60 sec pumping’ and the consequence ψ is ‘tank rupture’. Because 60 sec pumping must not immediately lead to tank rupture, the cause and consequence are related through a C-gate. If we want to prove the correctness condition $A(\varphi P \psi)$ we have to formalize the events φ and ψ .

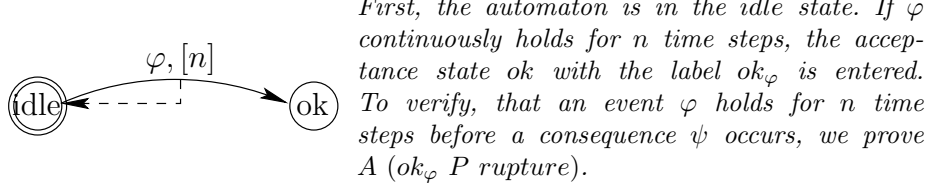
A first try is to formalize ‘tank rupture’ as $\psi := rupture$ and ‘60 sec pumping’ as $AG_{[60]}pumping$ (if *rupture* and *pumping* are the suitable atomic formulae). But this formalization is not adequate, because it does not require that ‘60 sec pumping’ is completed, before the consequence ‘rupture’ occurs. Assume a system state s , wherein *rupture* is true. The proof condition requires, that there exists a state s' before s , that fulfills the cause $AG_{[60]}pumping$. If $AG_{[60]}pumping$ holds for the predecessor of s , the cause and the consequence overlap, but the proof condition is fulfilled.

A second try is the formalization $A((G_{[60]}pumping) P X_{[60]}rupture)$. It ensures, that the consequence does not start until the cause has completed. But this is not a CTL formula. In CTL, every state operator must be prefixed with a path operator. So, we can formalize 60 sec pumping either as $\varphi := AG_{[60]}pumping$ or as $\varphi := EG_{[60]}pumping$. Again assuming a state s' before s , where *rupture* holds, the first formula states, that on all paths, starting at s' , pumping holds for 60 sec. This condition is too strong, because the cause must only occur on the path to s . The second formula states, that there exists a path starting from s' , where pumping holds for 60 sec, but this need not be the path leading to s . So we cannot formulate, that 60 sec pumping holds on *exactly* the path, that leads to rupture.

To solve this problem, we introduce so called observer automata. Such an automaton runs in parallel to the system model. It observes the event ‘pumping’ and enters an accepting end-state, when the 60 sec are over. This end-state flags, that 60 sec pumping have been occurred. Now, we can check, if this acceptance state was entered, before the corresponding consequence occurs. The entering of the acceptance state is a timeless event. Therefore we can prove, that it occurs on the path to and before the consequence.

We define a general observer automaton for accepting a temporal event as follows:

Definition 6 *observer automaton for temporal events*



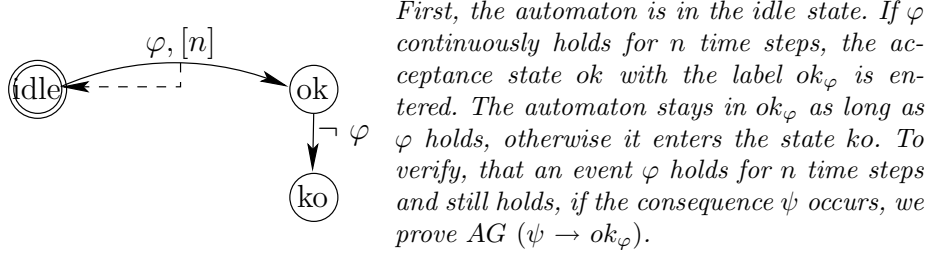
First, the automaton is in the idle state. If φ continuously holds for n time steps, the acceptance state ok with the label ok_φ is entered. To verify, that an event φ holds for n time steps before a consequence ψ occurs, we prove $A(ok_\varphi P \text{ rupture})$.

If we want to state ‘60 sec pumping’, we set $\varphi := \text{pumping}$ and $n := 60$. To verify, that the only cause for rupture is 60 sec pumping, we prove:

$$A(ok_{\text{pumping}} P \text{ rupture})$$

Observing typical fault tree events The automaton of definition 6 can observe the first fault tree event pattern of Górski (an event occurs for a certain time) and is the basis for the other two event patterns. For the second event pattern (event occurs and continuous to stay) we add an error state to the observer automaton which is entered, if the state ok is active, but the cause does no longer hold. Then we have to prove, that the ok state is active, when the consequence occurs, i.e. we get the proof condition $AG(\psi \rightarrow ok_\varphi)$.

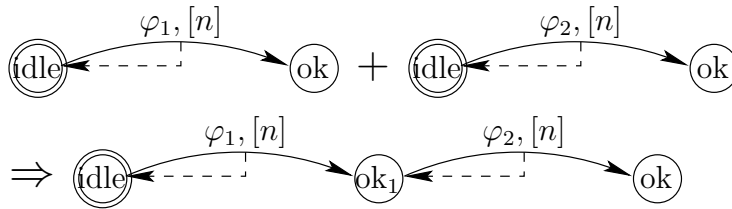
Definition 7 *observer automaton for ‘event continues to stay’*



First, the automaton is in the idle state. If φ continuously holds for n time steps, the acceptance state ok with the label ok_φ is entered. The automaton stays in ok_φ as long as φ holds, otherwise it enters the state ko . To verify, that an event φ holds for n time steps and still holds, if the consequence ψ occurs, we prove $AG(\psi \rightarrow ok_\varphi)$.

Finally, we can observe the third event pattern (sequence of temporal events) by sequentially composing different instances of the observer automaton of definition 6, corresponding to the sequence of events, and observe the acceptance state for the last event of this sequence.

Definition 8 *composing observer automata*



On the basis of observer automata for temporal events we can express every event pattern from Górski. These observer automata are run in parallel to the

system automata. Entering the acceptance state marks, that the entire temporal event occurred.

For formal arguments showing the correctness of this construction we refer to [12]. This paper presents an approach to extend CTL* and its subset CTL with the ability to specify quantitative timing properties using formulae of the Quantified Discrete-time Duration Calculus (QDDC) [13]. For the extended CTL, the resulting CTL[DC] formulae can be automatically proven by usual CTL model checkers. This approach also uses observer automata to describe the quantitative timing properties in QDDC formulae and checks, if the accepting end-state is reached. Correctness of this construction was proven in [12].

We derive the correctness of our approach by stating that the FTA event patterns can be formulated as QDDC formulae and the construction in [12] results in the described observer automata of definitions 6-8.

6 Validating the Pressure Tank FTA

In this section, we exemplify the approach of formal FTA with the pressure tank example from Sect. 2. After describing the formal model, we formalize the corresponding fault tree from Fig. 3, generate the proof conditions according to Fig. 4, and verify them with the model checker RAVEN [14]. The specification language of RAVEN are timed automata and it is able to check CCTL formulae. Therefore, the formal model are timed automata and the proof conditions are formulated in CCTL.

6.1 The Formal Model

The model depicted in Fig. 5 consists of eight single modules, representing the components of the pressure tank. The boolean variables c_1 and c_2 correspond to the electrical circuit c_1 and c_2 from Fig. 1 and show, if they are energized or not. The constants *timeout*, *fill_t*, *drain_t*, and *full_t* abbreviate the corresponding time intervals for the timeout, filling and draining time, and how long the pressure sensor can detect ‘full’ before the tank ruptures. Within this time span the control system has to react and to stop filling.

The module *env* describes the environment of the system, which indeterministically can press the push button s_1 (in state *env.c* the condition *env.press* holds). If pressed, the push button s_1 closes by entering state $s_1.c$ and stays therein as long as *press* holds. If *press* does no longer hold, the push button can indeterministically open by entering $s_1.o$ or stay in $s_1.c$ because of a defect. This behavior models the failure mode ‘fails to open’ of the push button s_1 .

The modules relay k_1 , relay k_2 , pressure switch s , and the pump *pump* are modeled analogously. The timer has similar behavior too, but stays closed for a maximum of *timeout* steps. Following the Fault Tree Handbook [23], we modeled the failure mode ‘fails to open’ for the relay k_2 , the push button s_1 , the timer *timer*, and the pressure switch s .

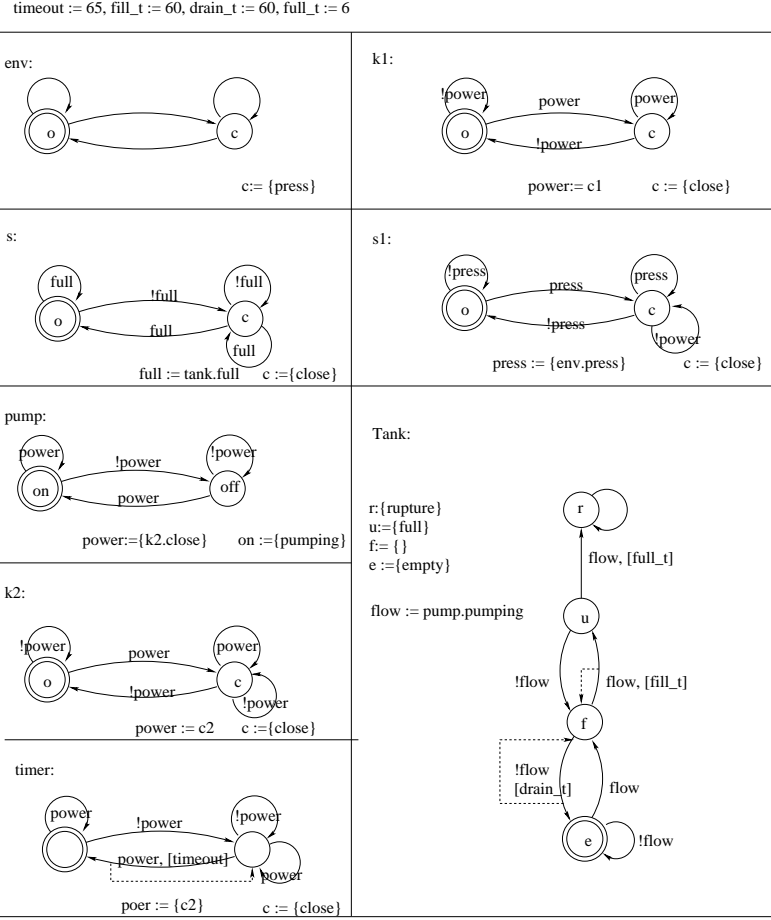


Fig. 5. Model of the pressure tank

The *tank* itself is either empty, filling, full, or ruptured. When *flow* ($flow := pump.pumping$) holds the state filling is entered. If the pump continues filling for $fill_t$ time units, the tank is full. The pressure switch detects if the tank is full and is fixed to the tank such, that the control system has enough time to stop filling, i.e. the tank ruptures not before $full_t$ time units *flow* in state full. If filling the tank is interrupted, the tank reaches state drain and empties in $drain_t$ time units.

We measure the time in seconds and set on time unit of the model to one second. The time values originate from the *Fault Tree Handbook* and we assign the value 60 to $fill_t$ and $drain_t$, the value 6 to $full_t$, and the value 65 to $timeout$ (see top of Fig. 5).

6.2 Formalizing the Fault Tree

For formal FTA, we have to formalize the fault tree from Fig. 3, i.e. the events have to be described in terms of the formal model and we have to decide, whether the gates from the informal analysis are decomposition or cause consequence gates.

The top event ‘rupture’ can easily be formalized as *tank.rupture*. The only cause for rupture is the continuous filling for more than 60 sec, which is formally $G_{[61]} \text{ pump.pumping}$. Now, we have to decide, if a decomposition or a cause consequence relation is described between the cause and the consequence. Because 60 sec pumping must not immediately lead to rupture the two events are connected through a cause consequence gate. The other gates in the fault tree are decomposition gates. The formalization of the whole fault tree is depicted in Fig. 6.

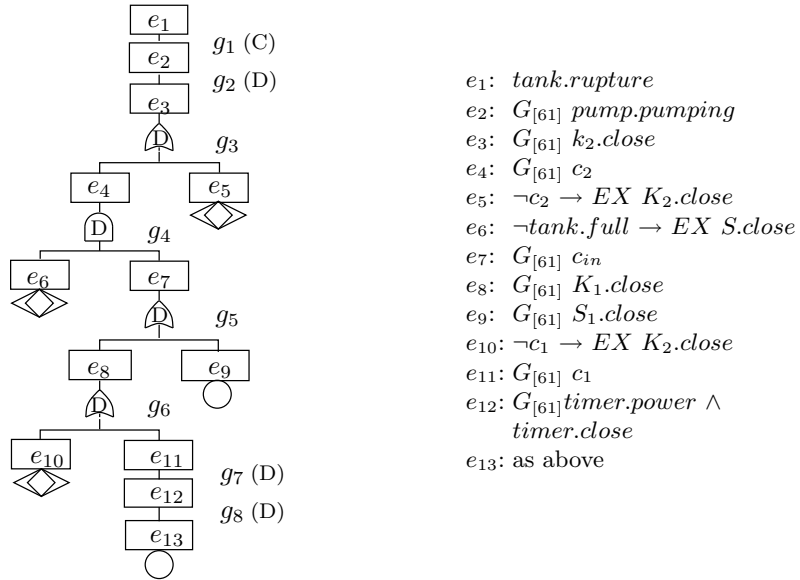


Fig. 6. Formalized Fault Tree

6.3 Checking Correctness and Completeness

Because *more than 60 sec pumping* is a temporal event, we have to add an observer automaton to the formal system model, which accepts this event. Therefore, we instantiate the automaton of definition 6 with $idle := idle_{pumping}$, $ok := ok_{pumping}$, $\varphi := pump.pumping$, and $n := 61$. The pattern of the completeness condition for a cause consequence gate is, according to Fig. 4, $A(\varphi P \psi)$. This

pattern is instantiated with $\varphi := ok_{\text{pumping}}$ for the cause and $\psi := tank.rupture$ for the consequence, resulting in

$$A (ok_{\text{pumping}} P tank.rupture)$$

the first verification condition for the completeness of the fault tree of Fig. 3. Analogous, we can derive the correctness condition

$$EF ok_{\text{pumping}} \rightarrow EF tank.rupture.$$

Both conditions are verified with the model checker RAVEN over the presented formal model.

The remaining gates are decomposition gates. The correctness condition is an implication from the cause to the consequence, the completeness an implication from the consequence to the cause. Because every event is temporal (more than 60 sec ...), we have to add corresponding observer automata for every event. Finally, we can prove the correctness and completeness of fault tree in RAVEN.

Theorem 1 *Correctness and completeness of the fault tree*

The fault tree of Fig. 6 is correct and complete, i.e. the proof conditions for every gate can be proven.

This theorem and the minimal cut set theorem (see Sect. 5.1) justify, that only the cut sets can cause the hazard ‘rupture’. All other reasons are ruled out.

6.4 Benefit of Model Checking Fault Trees

In the previous section, we presented the successful checking of the formal fault tree for the pressure tank example. But, the first model checking attempts failed. We had to fix syntax errors, flaws in the specification of the system model, and the formalizations of events.

The big advantage of model checking fault trees is, that incompleteness of fault trees by an omitted cause for the hazard is detected. We demonstrate this, by, hypothetically, ‘forgetting’ the cause ‘ K_2 fails to open’, the sub-event e_5 at gate g_3 in the fault tree of Fig. 6. When we try to prove completeness of gate g_3 , RAVEN generates the counterexample depicted in Fig. 7.

The system states are noted on the left and the horizontal lines on the right show, when the state is active (fat line) or not (thin line). The failure state is marked with the vertical line at the right. It states, that the relay K_2 is closed ($K_2.K_2\#0$) although S is open ($pr_switch.pr_switch\#0$) and therefore K_2 is de-energized. This situation describes that K_2 fails to open. This cause has to be added to the fault tree. So, the counterexample directly points to the forgotten cause.

7 Related Work

Hansen et al. [7] proposed a formalization of FTA in Duration Calculus (DC, [3]). She used FTA to derive formal specifications from safety requirements. The

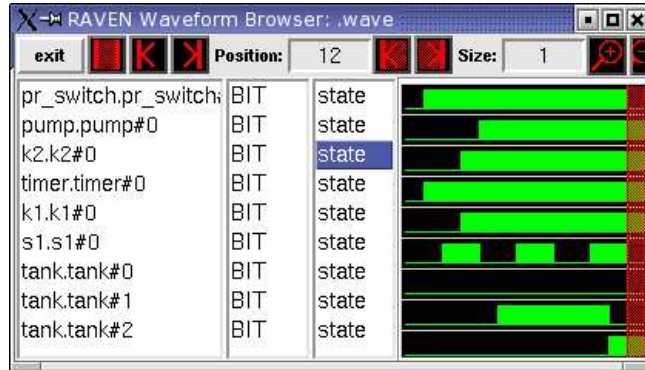


Fig. 7. Counterexample

formalization required, that the causes have to occur before (or at least at the same time as) the consequence. But, if causes and consequences are temporal events, the formalization allows an overlap, i.e. the (temporal) consequence can start before the (temporal) causes have finished. Furthermore, no minimal cut set theorem (see Sect. 5.1) was proven to justify the formalization.

Bruns and Anderson [2] presented a formalization for timeless events in the μ -calculus [8]. They distinguish between decomposition and cause consequence gates, as we do, but proved the minimal cut set theorem for decomposition gates only. If this formalization is used for temporal events, causes and consequences can overlap as well.

In [20] we discussed in detail the weaknesses of the two formalizations from above and presented an improved formalization in Interval Temporal Logic (ITL, [11]). We formalized decomposition and cause consequence gates and distinguished between synchronous and asynchronous AND-gates. The minimal cut set theorem was formally proven for this formalization for both, timeless and temporal events. Based on these results, we developed the CTL semantics for model checking FTA. The foundations of model checking FTA with the extension to timeless events by introducing observer automata and an example case study is the scope of this paper. In [22] we formally compared the three previous mentioned semantics with the CTL semantics. We proved in the specification and verification environment KIV [1], that if the fault tree consists of timeless events only, all four semantics are equivalent. The big advantage of the FTA semantics in CTL is, however, that it allows model checking the correctness and completeness of FTA.

A model checking approach is proposed in [19], as well. It uses the Duration Calculus model checker Moby/DC [21], which is based on phase automata and our ITL semantics from [20]. To prove correctness and completeness of the fault tree, the (negated) proof conditions for the gates have to be translated into phase automata and model checking excludes a common path with the system model.

This verifies, that the proof conditions hold. Because phase automata are not closed against negation, not every event can be translated into a corresponding phase automata, but patterns are given for the typical fault tree events from Górski [6].

8 Conclusion

We presented an approach for the tight integration of FTA with model checking, for analyzing high assurance software based systems like aerospace, nuclear power plants, transportation and, medical control. This approach combines a typical engineering safety analysis technique with a safety technique from software engineering. The benefit of of this combination is twofold. It provides the possibility of formally validating FTA by proving correctness and completeness of the fault tree. Second, defects and failures of components can be treated within the formal system model and, nevertheless, safety properties (no cut set implies no hazard) can be proven. This is the basis of qualitative and quantitative system assessment of formal models.

To automating the formal proofs by model checkers, we presented a FTA semantics in CTL. Automated proof support is a necessary precondition for the acceptance of new approaches in industry. This CTL semantics is adequate for timeless events, justified by the minimal cut set theorem. A reduction of temporal to timeless events by using observer automata allows to treat temporal events as well. The pressure tank case study exemplified the approach.

References

- [1] M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.
- [2] G. Bruns and S. Anderson. Validating safety models with fault trees. In J. Górski, editor, *SafeComp'93: 12th International Conference on Computer Safety, Reliability, and Security*, pages 21 – 30. Springer-Verlag, 1993.
- [3] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.
- [4] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, number 131 in LNCS. Springer, 1981.
- [5] Th. Filkorn, H.A. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE user's guide. Technical Report ZFE BT SE 1-SVE-1, Siemens AG, Corporate Research and Development, Munich, 1994.
- [6] J. Górski. Extending safety analysis techniques with formal semantics. In F. J. Redmill and T. Anderson, editors, *Technology and Assessment of Safety Critical Systems*, pages 147 – 163, London, 1994. Springer Verlag.
- [7] K. M. Hansen, A. P. Ravn, and V. Stavridou. From safety analysis to formal specification. ProCoS II document [ID/DTH KMH 1/1], Technical University of Denmark, 1994.

- [8] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 17(3):333–354, December 1983.
- [9] N. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.
- [10] K. L. McMillan. *Symbolic Model Checkings*. Kluwer Academic Publishers, 1990.
- [11] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
- [12] P. K. Pandya. Model checking CTL*[DC]. In T. Margaria and W. Yi, editors, *Proceedings of TACAS 2001*, LNCS 2031, Genova, Italy, 2001. Springer-Verlag Berlin Heidelberg.
- [13] P. K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. In *Proceedings of Workshop on Real-Time Tools RT-TOOL 2001*, Aalborg, Denmark, August 2001.
- [14] J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.
- [15] J. Ruf and T. Kropf. Modeling real-time systems with I/O-interval structures. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. Shaker Verlag, 1999.
- [16] J. Ruf and T. Kropf. Modeling real-time systems with I/O-interval structures. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 91–100. Shaker Verlag, March 1999.
- [17] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, 1997. IFIP WG 10.5, Chapman and Hall.
- [18] Jürgen Ruf and Thomas Kropf. Using MTBDDs for Composition and Model Checking of Real-Time Systems. In *FMCAD 1998*. Springer, November 1998.
- [19] Andreas Schäfer. Fehlerbaumanalyse und Model-Checking. Master’s thesis, Universität Oldenburg, 2001. in German.
- [20] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, CA, 2002.
- [21] J. Tapken. Model-checking of duration calculus specifications. Master’s thesis, University of Oldenburg, June 2001. <http://semantik.informatik.uni-oldenburg.de/projects/>.
- [22] A. Thums, G. Schellhorn, and W. Reif. Comparing fault tree semantics. In D. Haneberg, G. Schellhorn, and W. Reif, editors, *FM-TOOLS 2002*, Technical Report 2002-11, pages 25 – 32. Universität Augsburg, 2002.
- [23] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. Washington, D.C., 1981. NUREG-0492.