Part I

This is What a Part Would Look Like

Chapter 1

Modelling and Analysis of Component Faults and Reliability

Abstract: This chapter presents a process to design and validate models of reactive systems in the form of communicating timed automata. The models are extended with faults associated with probabilities of occurrence. This enables a fault tree analysis of the system using minimal cut sets that are automatically generated. The stochastic information on the faults is used to estimate the reliability of the fault affected system. The reliability is given with respect to properties of the system state space. We illustrate the process on a concrete example using the Uppaal model checker for validating the ideal system model and the fault modeling. Then the statistical version of the tool,Uppaal-SMC, is used find reliability estimates.

1.1 Introduction

Dependability of software systems in its widest meaning [3] is an area which calls for application of rigorous reasoning about programs. This is in particular the case for embedded software, where programs interact closely and continuously with a larger environment. Therefore, it has been investigated by developers of formal methods through decades. Here, Kaisa Sere with Elena Troubitsyna [24] have done seminal work which through the passed years has been continued with integration into development processes [27]. The work presented here has a similar perspective. It was inspired by model based testing of real-time systems [22] and analysis of service oriented home automation systems [12]. In these contexts, it is interesting to consider how models developed for testing or interaction analysis can be reused for safety analysis and perhaps even answer questions about the reliability of the overall system, because models are not inexpensive. It is a major development effort to build useful models of software and, even more so, of the context in which the software is embedded. The contribution in this paper is thus a systematic process, where behavior models are reused in safety analysis and reliability analysis of embedded systems.

The initial models have to describe the dynamics of a system, not only structural properties. They could come from Model Based Development (MBD) in the form of state charts or state machines for the software; for the environment there might be some form of tractable hybrid automata [16], perhaps in the form of timed automata [1] (TA). In other cases the models could come from model based testing or other analyzes. Whatever the origin, we assume that they describe the behavior of ideal, correct systems which are not affected by any faults.

Since the software is embedded in physical systems, faults will inevitably occur due to wear and tear. Software component failures may also be included. Although, they may often be hidden in electronic components with built-in intelligence. Faults are essentially events that affect the behavior of the system as a whole; they may be classified in many ways as described by Avizienis in particular, see [3] for further work. However, we assume that a list of likely faults is known for the system under consideration.

In a conventional process faults are used in a safety analysis without considering behaviors, but as pointed out in e.g. [15], faults are related to dynamics of a system. A further step is to integrate the analysis with behavioral models of a system. Here we are fortunate to be able to build on the work of Shäfer [23] who uses phase automata as the system model and the duration calculus to assign semantics to fault trees, and Thums et al. [26] who use TA for modeling and Computational Tree Logic (CTL) [13] for fault tree semantics. Finally, Bozzano et al. [9] have gone a step further and automated the

synthesis of fault tree from system modeled as Kripke structures, a result we extend to TA. Thus, there is a solid basis for model based safety analysis.

There is, however, a catch when augmenting an ideal model with faults and thus introducing failure modes. It should not be the case that faults provide desired functionality! This issue has been investigated by Liu and Joseph [20], who present suitable healthiness conditions. They are employed in a TA setting in [30] which is the formulation used here.

A system with failing components may be saved from failure by augmenting it with fault-tolerance mechanisms as done in [30]. Yet, this is costly, and there is still a probability of failure. The real question is: how reliable is the system? Here, a stochastic model is needed. By assigning probabilities to faults, an automata model becomes a Markov process, or if non-determinism is involved, a Markov Decision Process. Both can be handled with model checking techniques, see for instance [5, 14]. It may not be realistic to model check a larger model to get a figure for the reliability; so since the basic fault probabilities are estimates anyway, it is feasible to use the ideas of statistical hypothesis testing and get an answer with some chosen degree of likelihood. This is mechanized in Statistical Model Checking (SMC) [19] which we apply in this paper.

1.1.1 Overview

The systematic approach outlined above is presented succinctly in Section 1.2. In order to demonstrate the approach, we apply it to a concrete use case using the model checker Uppaal [8] and its statistical version Uppaal-SMC [11] in Section 1.3. Finally, we discuss limitations, related work and provide a conclusion and possible future work in Section 1.4.

1.2 A Development and Analysis Process

The development process motivated in the introduction has as objective to enable reliability assessment of reactive systems. The overall process, illustrated in Figure 1.1, is as follows:

1. Design and verify an *ideal model* of the system that correctly implements its requirements. As in any system modeling process, the models are derived from requirements and available components. Different types of requirements exist, from functional ones, that express the functionality that the system should provide to extra-functional ones, which for example constrain the time in which a function of the system should execute.

This kind of model is well known from model based development and forms a basis for verification of correctness of a system design.

2. Augment the *ideal model* with failure modes to produce a *faulty model* and verify that they invalidate some requirements.

   This is a novel step. It aims at analyzing requirements which are orthogonal to those for the ideal model. A borderline case is safety requirements prohibiting states that can cause harm to the larger environment of the system.

3. From the augmented model a fault tree for safety analysis can be derived. It enables to detect the weak points of the system and strengthen them if necessary.

4. Associate failure modes with failure rates to obtain a *probabilistic model*. It allows an assessment of system reliability. When validation fails, the previously generated fault tree can be used to determine if the component structure should be updated or its reliability improved. This is a novel step in conventional software development, but it is known from safety analysis. Since failures are stochastic in nature, analyzing them requires an estimate of the probability of their occurrence. Exactly how they are found is a gray area, ideally they are obtained through statistical experiments; but this requires testing many similar components. This is hardly feasible for complex components, so in practice they are most likely estimates based on experience. Nevertheless, it is assumed that Mean Time To Failures (MTTF) and failure rates are known in advance for both requirements and components.

This process is an iterative one, and failure in one of the validation steps implies a need to step back and modify the model or the properties specified, as illustrated by the dashed lines in Figure 1.1.

1.2.1 Ideal Model

Tractable models are usually finite state abstractions of the concrete system. Modeling the behavior of its components involves representation of their states, the transition between those, and their interactions with other components and possibly global state variables. The requirements are translated to predicates, usually in some dynamic logic in the case of reactive systems. The requirements are then checked to hold for the model. In general, two types of properties can be verified. Safety properties say that bad states are never entered, while liveness properties say that the system keeps on moving, in particular that it avoids deadlocks. Simulation is also of interest in this step, to observe the system evolution for a given period and ensure that it behaves as expected at least for the observed runs. In our setting it is an essential step in the process. Fault modeling does not make sense for an inherently faulty

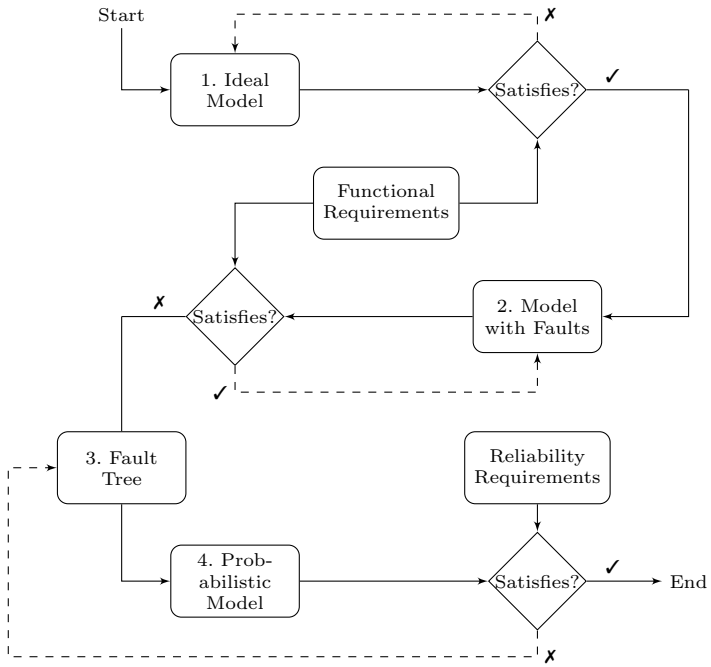


FIGURE 1.1: The process

system, or conversely it is hard to inject specific faults with any effect in a system that does not have to satisfy any properties.

1.2.2 Modeling Faults

We recall that a failure is a transition from the system providing correct service to incorrect service. The state of the system when delivering incorrect service is called an error state. Finally, the cause of an error is called a fault. When the system delivers correct service in the absence of faults, the model is augmented with component faults, represented as transitions from normal states to error states. For convenience, error states may be duplicated. The obtained model is called a Fault-Affected Automaton [30]. We recall its definition here.

Definition 1. (Fault-Affected Automaton) A fault-affected (or F-affected) automaton is an automaton with identified faulty transitions to error locations.

- $L \cup ERR$ is the union of the two finite and disjoint sets of normal and error locations.

- $l_0 \in L$ is the initial location, it is a normal state, thus the system starts correctly.

Fault transitions in the set F are the only transitions moving to error locations.

In order to ensure the correctness of the fault modeling, one needs to check the following healthiness condition. Given an F-affected automaton M of a correct automaton model S:

H 1. $M \backslash F \approx S$, meaning that when removing faulty transitions, the obtained model is bisimilar with the correct model.

Finally, for any $f \in F$, each fault-affected model $M \backslash \{F \backslash \{f\}\}$, should invalidate at least one of the properties of S. This is to ensure that each fault is significant in the model. Insignificant faults can be ruled out using Fault Tree Analysis, described in Section 1.2.3. Note that in some cases, a combination of faults leads to an error state, while their individual occurrence does not. In that case we would consider their combination as a fault in the set F, and we would need to check that the occurrence of this composite fault leads to a violation of a system property.

Note that although these conditions are formulated for automata, they can be interpreted for transition systems in general, see [20].

1.2.3 Fault Tree Analysis

Fault Tree Analysis (FTA) is used to determine the possible causes of a system failure, and enables one to identify critical components in a system architecture. It is a top down approach in which a top level event (TLE), representing a failure, is decomposed into simpler events composed by boolean connectives. These events can be further decomposed until a level of elementary events is reached. Boolean logic is then used to analyze the possible combinations of elementary events that can lead to the failure. The basic syntax of fault trees is composed of event symbols, representing top level-, intermediate-, or basic events, and logic gates that express boolean relations between events. Research in this area has provided better semantics for the syntax of fault trees, enabling the specification of event duration and sequencing [15] for example. With advances in modeling formalisms and tools, it has been made possible to ensure the correctness and completeness of FTA with regards to models decorated with faults [23, 26]. Recent research has also shown the possibility of automatic generation of fault trees from fault-affected models [9].

Here we show how to automatically generate fault trees with minimal cut sets using CTL. In order to generate fault trees we need to compute minimal cut sets. A cut set is a set of failures that lead to a TLE. A minimal cut set is a cut set reduced to include only necessary and sufficient failures for the TLE to occur.

For systems modeled as state machines, we define cut sets and minimal cut sets as follows. Given a set of faults F, an initial state I and a failure TLE,

Algorithm 1 Minimal Cut Sets

```
if IsCutSet(∅) then
  return ∅
end if
Waiting := {FS ∈ F | |FS| = 1}
mCS := ∅
while Waiting ≠ ∅ do
  for all FS ∈ Waiting do
    if IsCutSet(FS) then
      Waiting := Waiting \ FS
      mCS := mCS ∪ FS
    end if
  end for
  Waiting := PairwiseUnion(Waiting)
end while
return mCS
```

$CS \subseteq F$ is a cut set for TLE iff there exist a run of the system, starting from I, visiting all faulty states in CS before the failure state TLE. The cut set CS is minimal, iff there is no cut set CS' of TLE which is smaller, $CS' \subset CS$. Given $mCS = \{mCS_1, \cdots, mCS_n\}$ the set of minimal cut sets for a failure TLE and a set of faults F, we note that $mCS \subset \mathcal{P}(F)$.

It is possible to check if a set of faults FS is a cut set by checking if there exist a path where all faults in the cut set are active together with the failure, while faults outside of the cut set are not:

$$E\Diamond\ TLE \wedge FS \wedge \neg(F \setminus FS) \tag{1.1}$$

We recall that $E\Diamond\ \varphi$ means that for some paths in the model, there exist a state where φ holds.

To construct fault trees, Formula 1.1 can be used to decide the set of minimal cut sets for a given model, by iterating FS through $\mathcal{P}(F)$, starting from the sets with the smallest cardinality to ensure their minimality. This is realized by Algorithm 1.

The function $IsCutSet$ checks if its argument is a cut set using Formula 1.1. The algorithm starts by checking if the empty set is a cut set. If it is, then either the fault set F is incomplete or the TLE can be reached without any error being triggered (which probably indicates an issue with the model or the specifications). The algorithm explores the power set of F, checking for each element if it is a cut set. Since the power set is explored from the bottom, a cut set is minimal, once it is found. When a cut set is found, it is removed from the $Waiting$ set, as none of its supersets can be a minimal cut set. Once all sets in $Waiting$ have been checked, the $PairwiseUnion$ function is applied to it to move on to sets of higher cardinality.

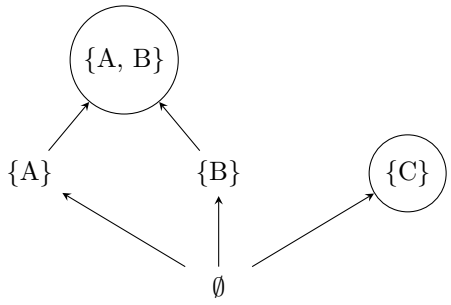


FIGURE 1.2: Example of Power Set Exploration

Figure 1.2 shows an example exploration of the set of faults $\{A, B, C\}$. The algorithm finds that C is a minimal cut set in itself, thus no other sets containing C are explored. A and B are not minimal cut sets. The pairwise union function joins them into the set $\{A, B\}$, which is found to be a minimal cut set. In this case the set of minimal cut sets is $mCS = \{\{A, B\}, \{C\}\}$.

In addition to identifying critical components and ruling out insignificant faults as already mentioned, FTA can be used to determine if components, or sets of components are in series or in parallel. Components in parallel will belong to one minimal cut set, while components in series will belong to separate minimal cut sets. This can be valuable as most representations of system models do not enable an easy visualization of this information.

1.2.4 Reliability Assessment

This step requires that the stochastic process is formulated with tractable distributions. The realism can always be discussed; but it is necessary to keep the model simple in order to get results. A component can fail in two ways, either temporarily (e.g. an unreliable communication channel) or permanently (e.g. a physical component breaking). Transient faults can be modeled using probabilistic branching, while permanent faults are modeled using probability distributions. Assuming constant failure rates, permanent failure transitions are modeled using an exponential distribution with parameter $\lambda = 1/MTTF$. This information is inserted into the model to determine the unreliability of the system–the probability that it fails after a given period.

Unreliability is expressed as a property over the global state space of the system. The probability of this property being verified is then estimated using statistical model checking (SMC), with two different possibilities. Firstly using hypothesis testing to validate that the probability of failure in a given time interval is less than a threshold, the time interval and threshold being part of the system requirements. Secondly using probability estimation to obtain an estimation of the system unreliability within a confidence interval.

1.3 Example

To illustrate the process, we apply it on an example system, simple enough so that models can be shown here and easily understood. We use UPPAAL and its statistical version UPPAAL-SMC to verify the model of the system and estimate its reliability. Note that UPPAAL and UPPAAL-SMC were chosen because they make it possible to apply the process on the same model, using model checking for verifying the correct modeling of faults in a first part and then using statistical model checking for evaluating reliability in a second part. We start by creating an ideal model of the example system.

1.3.1 Ideal Model

The system is a gas tank, shown in Figure 1.3. It is composed of five components:

- the tank structure,
- an input valve, controlling the incoming flow of gas in the tank,
- an output valve, externally controlled, providing gas,
- a sensor, measuring the level of gas in the tank,
- a controller, controlling the input valve based on the sensor's output.

Its function is to deliver gas when requested from its output valve. We assume this tank to have a capacity of 10L. When the gas level drops below 2L, the controller opens the input valve to refill the tank, until the level reaches 8L. If the level of the tank rises above 10L, it explodes. Obviously this is an undesirable event. Another requirement is that the tank should always be able to provide gas from its output valve, therefore it should never be empty. Now that we have specified the system and its requirements we continue to model it.

The first thing is to extract the variables from the specifications. We have:

- the level of the tank,
- the state of the sensor,
- the state of the input valve,
- the state of the output valve.

The declaration of these variables is shown in Listings 1.1. Constants are used to improve the clarity of the models. Note that the level of the tank and the sensor are initialized to a value that corresponds to a normal state of the

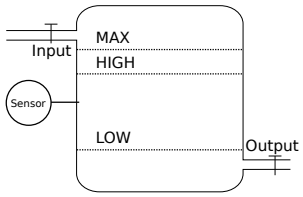


FIGURE 1.3: Gas Tank Example

system, where the level of the tank satisfies the requirements and the sensor reports a correct value.

Listing 1.1: Variable Definitions

```
const int MAX    = 10;
const int HIGH   = 8;
const int LOW    = 2;
const int CLOSED = 0;
const int OPEN   = 1;
const int INIT_LEVEL

const int MINUTE = 1;
const int HOUR = 60*MINUTE;
const int DAY = 24*HOUR;
const int YEAR = 365*DAY;

int level  = INIT_LEVEL;
int output = CLOSED;
int input  = OPEN;
int sensor = INIT_LEVEL;
```

Listing 1.2: Channel and Time Definition

```
broadcast chan levelSync;
broadcast chan stop;
broadcast chan open;
broadcast chan close;
broadcast chan updateSensor;
```

Listing 1.3: Level calculation

```
void calcLevel() {
  if (input == OPEN){
    level++;
  }
  if (output == OPEN){
    level--;
  }
  if(level < 0){
    level = 0;
  }
}
```

We detail the models shown in Figure 1.4. First, we use a *ticker* (Figure 1.4a) to discretize time and enforce a time unit among the models. The tank (Figure 1.4c) updates the level variable through the function calcLevel() shown in Listing 1.3. It also triggers the update of the sensor value. The sensor reports the level of the tank, and notifies of any changes. The input valve opens or closes when told to do so. At each time unit, the output valve can be opened or closed. Since its state is externally controlled, we create two probabilistic branches with equal weight to indicate that each time the transition is taken, there is an equal probability the valve is opened or closed. Finally, the controller implements the previously introduced specifications.

Note that components can synchronize and exchange information between each others using the channels listed in Listing 1.2. A "!" indicates that the automaton is initiating the synchronization, and possibly *sending a value*, while a "?" indicates that the automaton is waiting for synchronization, and possibly *receiving a value*. Note that it is required for SMC that all channels be broadcast, meaning that more than one automaton can receive a synchro-

P_1:	$A\Box\ \neg(level > MAX)$	✓
P_2:	$A\Box\ \neg(level = 0)$	✓
P_3:	$A\Box\ (level \geq LOW \wedge level \leq HIGH)$	✓

TABLE 1.1: Specifications expressed as UPPAAL properties

nization, and that a sender can synchronize even when no receiver is waiting for synchronization.

The model is verified against the system specifications; that it should not rupture, and should never be empty. We verify them with UPPAAL using the CTL formulas shown in Table 1.1. We recall that $A\Box\ \varphi$ means that for all paths and all state of the model, φ holds.

P_1 ensures that with the given design of the system, the tank cannot rupture. P_2 ensures that the tank cannot be empty. P_3 ensures that the control is satisfactory.

Theorem 1. The properties P_{1-3} are satisfied by the network of TA in Figure 1.4.

Proof. The properties are verified using UPPAAL. □

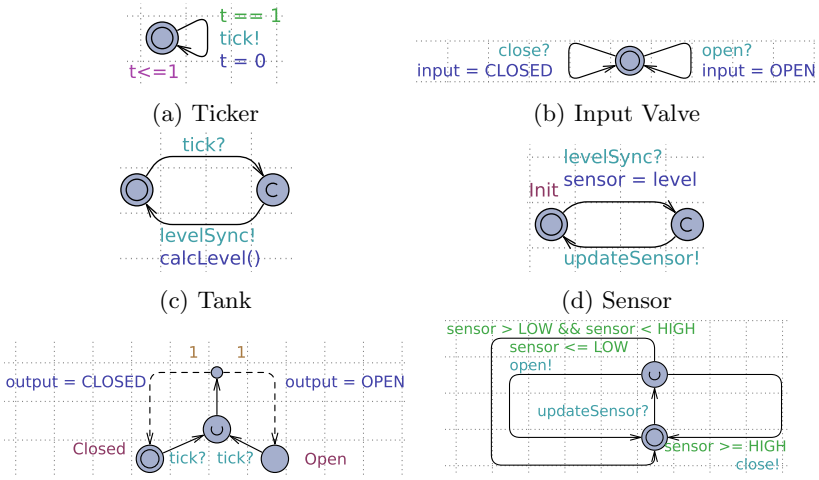
Other features of UPPAAL can be used to analyze the model. For instance the simulator can be used to get an indication of how the model evolves. The verifier can be used to track certain values for a set of simulations and visualize how they evolve. An example of this can be seen in Figure 1.5, where the value of `level` has been tracked for one simulation. This shows that the system evolves as expected.

1.3.2 Modeling Faults

In our example, we assume that only the input valve and the sensor can fail, and that their respective MTTF is 15 years and 20 years. We also consider that when the input fails, it stays open, and that when the sensor fails it stops reporting values, in effect stopping the controller. In order to keep the models clear, we propose to separate the modeling of the faults from the model of the components, as shown in Figure 1.6. Note that Figure 1.6e is not a fault model but an observer automaton. This is used to make queries more clear by using the *rupture* variable rather than $level > MAX$. Note also that the use of the *id* parameter in the $sensorFail[id]$ synchronization is in anticipation to the instantiation of several sensor in a later step.

Theorem 2. None of the properties P_{1-3} are satisfied by the fault affected model of the system.

Proof. Counter examples are found using UPPAAL. □



(a) Ticker (b) Input Valve

(c) Tank (d) Sensor

(e) Output Valve (f) Controller

FIGURE 1.4: Models of the System Components

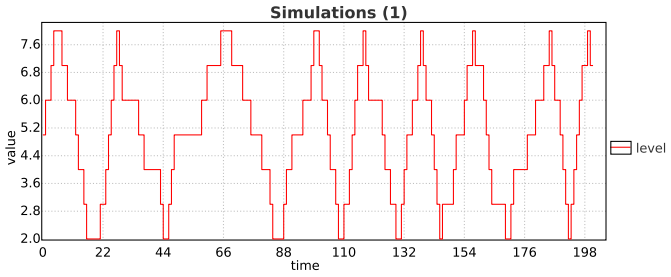


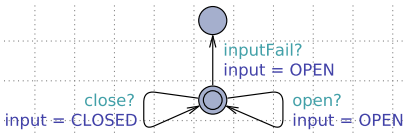
FIGURE 1.5: Simulation

Note that we have assumed only permanent faults and simple failure models. However, transient faults can also be modeled using probabilistic branching as in Figure 1.7a which shows a model of an unreliable sensor. Figure 1.7b shows a complex observer that models a failure occurring when two out of ten measurements are erroneous. We however do not use these model in our analysis to keep the example simple.

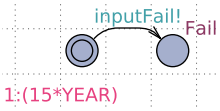
Having a fault decorated model, we move on to FTA to obtain an overview of the combinations of faults leading to failure of the system.

1.3.3 Fault Tree Analysis

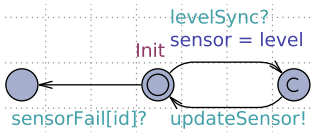
We construct the fault tree using the FTA explained in Section 1.2.3. We take as TLE the rupture of the tank. The queries run by the algorithm are shown in Table 1.2. Using this analysis, we can observe that both the input



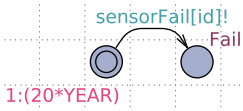
(a) Input with Faulty Transition



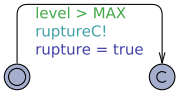
(b) Input Failure



(c) Sensor with Faulty Transition

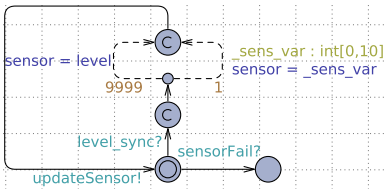


(d) Sensor Failure

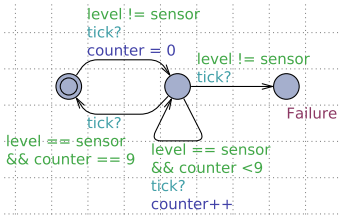


(e) Monitor for tank rupture

FIGURE 1.6: Fault-Affected Component Models



(a) Unreliable Sensor



(b) Complex Failure Observer

FIGURE 1.7: Unreliable Sensor and Associated Complex Failure Observer

valve and the sensor are single points of failure of the system. Single points of failure are usually not desirable, but can be acceptable if they are highly reliable. In order to estimate this we assess the reliability of the system.

1.3.4 Reliability Assessment

The example system can fail in two different ways. Either the tank ruptures or it becomes empty. Both are considered reliability issues, since they prevent normal operation. Only tank rupture is considered a safety issue, considering that it may endanger lives. These two cases are captured by the specifications

$E\Diamond\ rupture \wedge \neg Input.Fail \wedge Sensor.Fail$	✓
$E\Diamond\ rupture \wedge Input.Fail \wedge \neg Sensor.Fail$	✓

TABLE 1.2: Output of the FTA

of properties P_1 and P_2 in Table 1.1. In order to conduct the analysis, we need to set up reliability and safety requirements. We specify the followings:

1. the probability of system failure within one year should be lower than 10%,
2. the probability of catastrophic event within three years should be lower than 5%.

A system failure corresponds to a transition from a state in which the system delivers correct service to one where it does not. A catastrophic event is a failure of the system that impacts the system environment, in this case the rupture of the tank. Note that a catastrophic event implies a system failure, while the opposite is not true.

We start by evaluating unreliability within one year of service. We thus ask the question,

$$\textit{“What is the probability } \mathbb{P}_M(\Diamond_{t\leq 1year}(rupture \vee level = 0))\textit{?”},$$

assuming a time unit in minutes. We consider that we want an uncertainty of $\varepsilon = 0.03$ and that we want a 95% confidence that the result is correct. Therefore we set the significance level to $\alpha = 0.05$.

With a time of 31 minutes[1], we obtain an approximation interval of $[0.0924592, 0.152456]$, which goes above the required 10%. We can therefore not guarantee that the model satisfies the requirements.

We then assess the safety of the system; that the probability of catastrophic event during a three years period is less than 5%. Given the extended time period, we use hypothesis testing, that requires a lower number of simulations than probability evaluation. We thus ask the question, given the fault affected model M of the system,

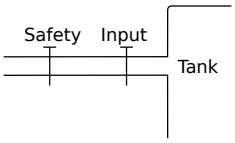
$$\textit{“is } \mathbb{P}_M(\Diamond_{t\leq 3year} rupture) \geq 0.05\textit{?”},$$

assuming a time unit in minutes. We set probabilistic deviations of $\delta = \pm 0.001$ and the probabilities of Type I and II errors of $\alpha = \beta = 0.05$. With a verification time of 2 hours and 24 minutes we get a positive answer, indicating that the safety requirement of the system is *not* met.

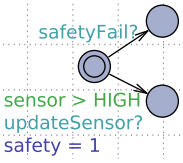
Note that we use minutes as the time unit as the number of simulations required using seconds makes the estimation process excessively long. This is obviously a drawback of SMC, since the execution time depends heavily on the length of each trace. However, compared to analytical solutions this is expected to scale better, since the generation of traces can be parallelized.

The probability of failure is too high w.r.t. the specifications. We need to strengthen the critical points of the system revealed by the FTA in order to obtain an acceptable reliability. To do so, different possibilities are available.

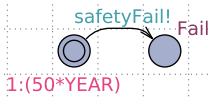
[1] All experiments are run on an i7 quad core 2.10GHz laptop with 8GB of RAM.



(a) Input pipe upgraded with a safety valve



(b) Safety Valve Model



(c) Safety Valve Failure Model

FIGURE 1.8: Safety valve

The easiest would be to increase the reliability of the individual components. However, this is not always possible, due to cost or physical constraints. Another is to add additional safety or redundant components to the system. In this case we add a safety valve to the input and a redundant sensor.

The safety valve closes the input pipes when the tank level exceeds the level $HIGH$. This safety valve is directly connected to the sensor, and as it is simpler than the input valve, we consider its reliability to be higher (MTTF of 50 years). The safety valve, its model, and failure model are shown in Figure 1.8. The function calcLevel() of Listing 1.3 is updated to take it into account by constraining the increase of the level of the tank to when the safety valve is open.

The redundant sensor is introduced in the system by instantiating a new Sensor and SensorFail process in the model.

After introducing these additional components, the FTA is performed again. Its outputs indicates that the fault tree is composed of two minimal cut sets, one containing the failures of the input and safety valves, the other the failures of the two sensors. The absence of minimal cut set with a single components indicates the absence of single point of failure.

The statistical estimation of system reliability and safety is then performed again. We first obtain an estimation of reliability within the interval $[0.0321459, 0.0919823]$ after 24 minutes. In order to convince ourselves of the results, we use hypothesis testing that, after 39 minutes confirms the result. We are thus 95% confident that the unreliability of the system within a 1 year period is less than 10%.

The hypothesis testing for the system testing with the updated model results in a negative answer after 11 hours and 28 minutes. We thus have obtained a satisfactory model of the system with regards to its specifications.

1.4 Discussion, Conclusion, Related and Further Work

1.4.1 Discussion

The process presented in this paper relies on TA and model checking. Model checking raises the issue of the scalability of the process, due to the state space explosion problem. This is a well known problem and abstraction and optimization techniques can be used to reduce the state space and render model checking feasible. The choice of Uppaal as the tool for model checking also reduces verification time as it seems to be the most efficient tool for model checking TA [29]. The second question regarding scalability is about the statistical model checking of the TA augmented with probabilistic transitions. The complexity of statistical model checking depends on the length of the traces to be generated, and the confidence requested for the statistical results. We have seen this in our experiments, where the combination of high time granularity and large time intervals make statistical model checking time consuming.

Regarding the choice of the tool, we note that Uppaal-SMC is said to perform better [11] than the PRISM [18] tool for statistical model checking. However, PRISM also enables probabilistic model checking, which can be more efficient than SMC when applicable. We could thus imagine modeling the system and faults in Uppaal and use model checking to ensure the correctness of the modeling, and perform probabilistic model checking when feasible using PRISM.

1.4.2 Conclusion

In this chapter, we presented a process to design and validate systems modeled as network of TA. The models are extended with fault transitions to error states that can be used in the application of two novel steps. We first showed how to perform an FTA based on these extended models, that helps identifying single points of failure and ensuring the correctness of the models. An algorithm to generate complete and correct fault trees with minimal cut sets was presented to facilitate FTA. The second novel step showed how to augment the fault decorated models with probabilities to perform reliability analysis of the system. We finally illustrated this process on a example, using Uppaal and Uppaal-SMC as tool support. Being based on model checking, the process is inherently limited in terms of the size of the system to be analyzed. However, this is a well known problem, and techniques for abstraction, reduction and simplification are available to help reducing it.

1.4.3 Related Work

Reliability and safety are broad areas of research that have focused the attention of many researchers.

Regarding relation between formal models and FTA, we mention the work of Sere and Troubitsyna [24] who use FTA to refine formal specifications written with the action system formalism [4]. The difference with our work is that the fault tree is used to refine the system model, while we derive it from the system model and use it as to validate and analyze the model.

Assessing safety through linking system reliability and components reliability using probabilistic models is also the objective of the work of McIver et al. [21]. The link is made through establishing probabilistic data refinement by simulation and is limited to sequential models. Troubitsyna takes this work further by using it in combination with the action system formalism, enabling its application to reactive systems in [28].

Regarding reliability analysis based on formal stochastic system models, we mention the work of Kwiatkowska et al. [17] using the PRISM tool [18]. Here our work emphasizes more the analysis process than the tool usability.

PRISM is also used by Tarasyuk et al. [25] to introduce reliability assessment in the Event-B refinement process. Here our work differs in that the process emphasizes fault modeling and incorporates FTA for identifying critical components.

Another use of statistical model checking in the context of reliability and safety analysis is shown by Arnold et al. [2] with the DFTCalc tool. This tool focuses on FTA to compute system reliability and MTTF. The difference in their work is that the fault tree serves as the basis of the analysis while we derive it from a formal model of the system. The advantage of DFTCalc is to have a more expressive syntax for fault tree (SPARE and Priority AND gates), but the correctness and completeness of fault trees cannot be checked. Moreover, using formal models and Uppaal also allows for checking safety and liveness properties.

Bozzano et al. [9] present a set of algorithmic strategies that enable the generation of a fault tree with minimal cut sets. The algorithms are designed for systems modeled as Kripke structures, and are implemented in the FASP/NuSMV-SA safety analysis platform [10]. The algorithm we propose rely only on the use of reachability queries. We also mention [6, 7] who propose to use retrenchment technique for modeling faults. A (concrete) faulty system is thus related to an (abstract) ideal system via a retrenchment relation. They then present algorithms for generating resolution trees first for timeless acyclic combinational circuits, and then for cyclic combinational circuits with clocks. Resolution trees can then be transformed into conventional fault trees, with the advantage that they provide more detailed relations between the faults, compared to the fault tree generated by our algorithm that only contain minimal cut sets.

1.4.4 Further Work

To make the process more useful and interesting for safety and reliability analysis, improving the generation of fault trees is an essential point. As already mentioned, the fault trees generated by our algorithm are *flat*, in the sense that they only provide minimal cut sets. It is however important to be able to visualize the nested relations between the faults, and generating nested trees would provide more insights. Generated such trees using a combinatorial approach is possible, but would not scale well. Applying on-the-fly algorithms such as the ones used in [9] would improve the performance, and should be investigated.

Another challenge is to make the process easier to use for practitioners in the fields of reliability and safety analysis. While preparing this paper, we have recognized two direct enhancements to UPPAAL to improve its applicability in this area. The first is to enable the specification of faults in the UPPAAL GUI and the second is to implement Algorithm 1 in UPPAAL.

Currently, UPPAAL does not provide any contextual understanding of TA, their states or transitions. Adding such contexts, by for example differentiating between system and environment models, normal and erroneous states, could facilitate modelling. Moreover, adding a notion of faults and fault affected models would enable the automation of several steps of the process presented here. The tool could for example automatically verify that requirements are met by the *ideal model*, and that each specified fault invalidates at least one of them.

With a specified set of faults and failures, Algorithm 1 could be implemented in the tool and generate a visual representation of the fault tree. The algorithm can currently be executed using the command-line interface to the UPPAAL verifier, but including it in the GUI would increase its usability. The reliability assessment could also be automated, and we are working on an extension that calculates the probability for each minimal cut set to trigger the TLE. This way the most likely path to a failure can be determined.

Adding features such as these will make UPPAAL more usable for practitioners and support the use of formal methods in industrial applications.

Bibliography

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *TCS*, 126(2):183 – 235, 1994.

[2] Florian Arnold, Axel Belinfante, Freark Van der Berg, Dennis Guck, and Marille Stoelinga. DFTCalc: A tool for efficient fault tree analysis. In Friedemann Bitsch, Jrmie Guiochet, and Mohamed Kaniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 293–301. Springer Berlin Heidelberg, 2013.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[4] R.J.R. Back and K. Sere. From action systems to modular systems. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 1994.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[6] Richard Banach and Marco Bozzano. The mechanical generation of fault trees for reactive systems via retrenchment I: combinational circuits. *Formal Aspects of Computing*, 25(4):573–607, 2013.

[7] Richard Banach and Marco Bozzano. The mechanical generation of fault trees for reactive systems via retrenchment II: clocked and feedback circuits. *Formal Aspects of Computing*, 25(4):609–657, 2013.

[8] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[9] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In *Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 2007.

[10] Marco Bozzano and Adolfo Villafiorita. The FSAP/NuSMV-SA safety analysis platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[11] Peter Bulychev, Alexandre David, Kim G. Larsen, Marius Mikučionis, Danny Bøgsted Poulsen, Axel Legay, and Zheng Wang. UPPAAL-SMC: Statistical model checking for priced timed automata. In Herbert Wiklicky and Mieke Massink, editors, *Quantitative Aspects of Programming Languages and Systems,* Proceedings of the 10th Workshop on, volume 85 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16. Open Publishing Association, 2012.

[12] Peter H. Dalsgaard, Thibaut Le Guilly, Daniel Middelhede, Petur Olsen, Thomas Pedersen, Anders P. Ravn, and Arne Skou. A toolchain for home automation controller development. In *39th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2013, Santander, Spain, September 4-6, 2013*, pages 122–129, 2013.

[13] E.Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.

[14] Kousha Etessami, Marta Z. Kwiatkowska, Moshe Y. Vardi, and Mihalis Yannakakis. Multi-objective model checking of markov decision processes. *Logical Methods in Computer Science*, 4(4), 2008.

[15] Kirsten M. Hansen, Anders P. Ravn, and Victoria Stavridou. From safety analysis to software requirements. *Software Engineering, IEEE Transactions on*, 24(7):573–584, Jul 1998.

[16] T. A. Henzinger. The theory of hybrid automata. In *LICS 1996*, pages 278–292. IEEE, 1996.

[17] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Eval. Rev.*, 36(4):40–45, 2009.

[18] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591, 2011.

[19] Kim Guldstrand Larsen and Axel Legay. Statistical model checking past, present, and future - (track introduction). In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pages 135–142, 2014.

[20] Zhiming Liu and Mathai Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst.*, 21(1):46–89, 1999.

[21] Annabelle McIver, Carroll Morgan, and Elena Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. In *IRW/FMP 98,* Proceedings of, 1998.

[22] Petur Olsen, Johan Foederer, and Jan Tretmans. Model-based testing of industrial transformational systems. In *Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*, pages 131–145, 2011.

[23] Andreas Schäfer. Combining real-time model-checking and fault tree analysis. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 522–541. Springer Berlin Heidelberg, 2003.

[24] Kaisa Sere and Elena Troubitsyna. Safety analysis in formal specication. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM99 Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1564–1583. Springer Berlin Heidelberg, 1999.

[25] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. From formal specification in Event-B to probabilistic reliability assessment. In *Dependability (DEPEND), 2010 Third International Conference on*, pages 24–31, July 2010.

[26] Andreas Thums and Gerhard Schellhorn. Model checking FTA. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 739–757. Springer Berlin Heidelberg, 2003.

[27] Elena Troubitsyna. Dependability-explicit engineering with Event-B: Overview of recent achievements. *CoRR*, abs/1210.7032, 2012.

[28] Elena A. Troubitsyna. Reliability assessment through probabilistic refinement. *Nordic Journal of Computing*, 6(3):320–342, 1999.

[29] Farn Wang. Efficient verification of timed automata with bdd-like data structures. *International Journal on Software Tools for Technology Transfer*, 6(1):77–97, 2004.

[30] Miaomiao Zhang, Zhiming Liu, Charles Morisset, and Anders P. Ravn. Design and verification of fault-tolerant components. In *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 57–84. Springer Berlin Heidelberg, 2009.