Abstract—Embedded real-time systems are increasingly ap-
plied in safety-critical environments like cars or aircrafts. Even though the system design might be free from flaws, hazardous situations may still be caused at run-time by random faults due to the wear of physical components. Hazard analysis is based on fault trees or failure propagation models. These models are created at least partly manually. They are usually independent from the software models which are used for checking safety and liveness properties to avoid systematic faults. This is particularly bad in cases, where the software model contains manually specified operations to deal with random faults which have been identified by hazard analysis. These operations include replacing the faulty components by reconfiguration. We propose to generate a failure propagation model automatically from the software model to check whether the results of hazard analysis have been properly accounted in the specification of reconfiguration operations. In contrast to other approaches, our approach considers the real-time properties of the system and adds explicit failure propagation times based on using timed automata for model specification.

I. INTRODUCTION

Embedded real-time systems are increasingly interacting with the physical world. They are usually software intensive, i.e., their functionality and correctness depend on high quality software. As they are often employed in safety-critical contexts, guaranteeing this high quality software becomes an absolute must. Besides testing and simulation, formal verification of a software model and automatic code generation have become an accepted approach to improve software quality and guarantee correctness.

Formal verification enables, in particular, checking safety and liveness properties of the system under development. This prevents systematic faults caused by malfunctioning software. However, due to the interaction with the physical world, errors in such systems may also be caused by random faults1 that may occur, for example, due to the wear of physical components.

Dealing with random faults caused by malfunctioning hardware is based on hazard analysis [12], [16]. This analysis checks possible combinations of hardware faults that lead to a hazard and computes the probability of the hazard’s occurrence. The system developer uses this information to implement the system and, in particular, its software such that the probability of the occurrence of a hazard is acceptable. This means, the software guarantees that the hazard only occurs with a certain probability [16].

Self-healing may be used to reduce the occurrence probability of hazards at runtime. Self-healing systems react to observed faults autonomously such that the system is returned into a safe state [15]. The effectiveness of this self-healing operation needs, of course, to be analyzed. In particular, we need to analyze whether the self-healing is executed fast enough to prevent failures from causing a hazard.

We presented such an analysis in [14]. This analysis operates on Timed Failure Propagation Graphs (TFPG). TFPGs are failure propagation models extended by propagation times. The benefit of TFPGs is their minimality concerning the information needed for the analysis of self-healing operations. The propagation times are needed to analyze whether the self-healing operation is executed fast enough.

However, the manual construction of failure propagation models without timing information is already a tedious and error prone task. Some approaches [13], [10] generate failure propagation models from behavior models which already exist for the system. The underlying idea of these approaches is to compare the reachable behavior of a system or part of the system with the behavior under the injection of failures. Based on this information, failure propagation models are constructed guided by the rules introduced in [17]. However, these approaches do not consider propagation times.

In this paper, we present an approach for the automatic generation of timed failure propagation graphs (TFPG) from timed automata [1], [4]. TFPGs are failure propagation models [7] annotated with the propagation times of failures. The benefit of TFPGs is their minimality concerning the information needed for hazard analysis. We adapt the idea of the existing approaches and extend it by the computation of propagation times to generate TFPGs. We adjust the evaluation of the behavior under injected failures to our behavior models. The propagation times of TFPGs are then computed from the time information needed for the analysis of self-healing operations. The propagation times of TFPGs are then computed from the time information needed for the analysis of self-healing operations.

In the next section, we introduce our models of the system structure and behavior. We then present TFPGs in Section III. The description of the generation of TFPGs from the system behavior model follows in Section IV. Section V discusses related work before we conclude the paper in Section VI.

II. SYSTEM MODEL

A. System Architecture

Since hazard analysis is based on the propagation of failures through system components as described in [14], we

1According to Laprie et al. [2], an error is the deviation from a correct system state. A fault is the cause of an error.
model the system by means of components. In the course of this paper, we use the component model of Mechatronic-UML [3], [5]. The component model distinguishes between component types and component instances. A component instance is the occurrence of a component type. Component instances of one component type have the same behavior specification but may be in different system states. The component model allows for multiple instantiation of components within a system.

Components communicate via ports using messages. Component types and port types are instantiated in a component instance configuration. The component instances are connected via their port instances using connectors that specify additional behavior for modeling delays of messages. We use deployment diagrams to model hardware nodes and the connection of component instances and hardware nodes.

![Deployment Diagram](image)

Fig. 1. Example of a deployment

Figure 1 shows an example of a deployment. It contains three component instances c1, c2, and c3 of types C1, C2, and C3, respectively. The component instances are illustrated by rectangles. The deployment further contains the two hardware nodes h1 and h2 depicted by boxes. c1 and c3 for example are connected via their ports p5 and p6. The ports are directed such that messages may only flow in the direction indicated by the triangles.

### B. Real-time Behavior

We model the behavior of component types and instances by timed automata as defined in [4]. A timed automaton is a finite automaton that is extended by a set of real-valued variables called clocks and has locations instead of states. A location represents a set of states. By using timed automata, the developer defines time-dependent behavior. Thus, the behavior of the component does not only depend on its inputs but also on the point in time when these inputs are received. This is essential for modeling real-time systems.

Based on the clocks, a timed automaton specifies time guards, clock resets, and invariants. A time guard is a clock constraint that restricts the execution of a transition to a specific time interval. A clock reset sets the value of a clock back to 0 when a transition is fired. Invariants are clock constraints associated with locations that forbid that a timed automaton stays in a location when the clock values exceed the bounds of the invariant.

Figure 2 shows an example of a timed automaton. The timed automaton consists of eight locations l0 to l7 and eleven transitions connecting the locations. The invariant c1 < 15 of location l0 specifies that l0 may only be active while the value of c1 is less than 15. Accordingly, the time guard 5 ≤ c1 ≤ 10 restricts the firing of the transition from l0 to l1 to the respective time interval. The time intervals are interpreted with respect to the values of the clock c1 and not with respect to the global time that has passed since the system has been started. The reset at the transition from l1 to l3 sets the clock c1 back to 0.

In addition to time guards and resets, a transition may carry an action symbol that specifies an input action or output action of the timed automaton. Input actions are suffixed by "?", output actions by "!". In order to relate actions to ports of components, we prefix actions by the name of the port. We thus use actions of the form p.a. p specifies the port name and a the action name. In our example, the input action p1.i1 specifies that input i1 is received via port p1. The output action p3.o2! specifies that the output o2 is sent via port p3.

### C. Component Context

To analyze the behavior of the timed automaton, we need to provide an environment model that sends the inputs of the timed automaton and receives its outputs as illustrated in Fig. 3. We call this environment the context of the timed automaton and its corresponding component. The context specifies timed automata for all inputs (input behavior) and outputs (output behavior) of the timed automaton of a component type.
timed automata consisting of the component’s timed automaton and its context is denoted as the \textit{component-and-context network} (CCN).

We need to analyze the reachable behavior of a CCN to construct TFPGs. The reachable behavior of the CCN is defined by a \textit{zone graph} [1], [4] which contains all run-time states that the CCN may visit during its execution.

III. TIMED FAILURE PROPAGATION GRAPHS

TFPGs model the propagation of failures through the system and particularly specify the propagation of failures over time. A TFPG does not reflect the paths of the reachable behavior, but it only relates incoming failures of a component to the component’s outgoing failures. Thus, the TFPG is a compact representation of the behavior of a component under the influence of failures.

Our approach follows the terminology of Laprie et al. [2]. \textit{Failures} are the externally visible deviation from the component’s behavior. They are associated with ports where the component instances interact with their environment. \textit{Errors} are the manifestation of a \textit{fault} in the state of a component, whereas a fault is the cause of an error. Errors are restricted to the internals of hardware nodes.

For the generation of TFPGs from timed automata, we restrict ourselves to failures. This is due to the fact that in our system model timed automata are only specified for software components. But random errors only occur in hardware nodes and become failures when they propagate to software components.

Failures are classified using a failure classification like the one by Fenelon et al. [6]. We distinguish the failure classes \textit{service}, \textit{value}, \textit{early timing}, and \textit{late timing}. A value failure specifies that a value deviates from a correct value, e.g., an erroneous action at the port of a component. A service failure specifies that no value is present at all, e.g., a component crashed and does not output any values. A timing failure specifies that a message has been delivered outside of a defined time interval, e.g., too late or too early.

TFPGs relate an outgoing failure to a set of combinations of incoming failures. In TFPGs, failures are represented by rectangles labeled with the according failure variable. Operators are represented by circles labeled with the according logical operator. Edges are labeled with time intervals that specify the minimum and maximum propagation time that a failure needs to propagate from the edge’s source to the edge’s target.

For a component \(k\), a port \(p\), a failure class \(fc\), and a direction \(d\), we create failure variables that are named according to the following schema: \(f_{k,p,fc}^d\) with \(d \in \{i, o\}\) and \(fc \in \{v, s, e, l\}\). \(i\) and \(o\) specify the direction of failures – \(i\) stands for incoming and \(o\) for outgoing. For failure classes, we use the following symbols: \(v\) for value failure, \(s\) for service failure, \(e\) for early timing failure, and \(l\) for late timing failure. For a component instance configuration, we instantiate failure variables by instantiating components. The notation described above holds for component instances and port instances analogously.

All nodes with in-degree zero are labeled with incoming failure variables. All nodes with out-degree zero are labeled with outgoing failure variables. All other nodes are labeled with a logical operator, i.e., either \& or \(\geq\) 1. In the remainder, we write “AND-node” for nodes labeled with \& and “OR-node” for nodes labeled with \(\geq\) 1.

![Fig. 4. Timed Failure Propagation Graph](image)

Figure 4 shows a TFPG that relates the outgoing value failure \(f_{c1,p3,v}^o\) at port \(p3\) of component \(c1\) to the joint occurrence of the incoming value failures \(f_{c1,p1,v}^i\) and \(f_{c1,p2,v}^i\) at ports \(p1\) and \(p2\) of component \(c1\). The interval at the edge from \(f_{c1,p1,v}^i\) to the \&-node specifies, that a failure needs at minimum 10 and at maximum 20 time units to be propagated from failure variable \(f_{c1,p1,v}^i\) to failure variable \(f_{c1,p3,v}^o\).

The edge originating from the AND-node has a propagation time interval of \([0, 0]\). This means, beyond this node, failure propagate in zero time. During TFPG generation, we construct the TFPG in this way, because we do not know how to divide the propagation time interval. We therefore, label the edges originating from the incoming failure with the propagation time intervals. All other edges are labeled with \([0, 0]\). This does not affect the correctness of the computed propagation timed during the analysis [14].

IV. GENERATING TIMED FAILURE PROPAGATION GRAPHS

In this section, we describe the generation of TFPGs for the timed automaton of a component type. The key idea is to determine which incoming failures cause which outgoing failures and how long an incoming failure needs to cause an outgoing failure.

The process for the generation of the TFPGs for a component type is shown in the enumeration below. All steps are conducted completely automatically.

1) Compute the reachable behavior of the component type with correct inputs.
2) Compute the reachable behavior of the component type with injected failures.
3) Construct the TFPGs from the reachable behavior with injected failures.

In \textbf{Step 1}, we determine the reachable behavior of the timed automaton for the case that no failures are injected into the component. This is done with the help of a CCN that consists of the timed automaton the \textit{initial context}. The initial context consists of a set of timed automata that send and receive all input and output actions of the timed automaton without time constraints. The initial context is generated from the timed automaton of the component type.
In Step 2, we create a refined context for the component automaton. The refined context restricts the sending and receiving of actions to the time intervals at which the component automaton receives and sends these actions. It is constructed from the zone graph of Step 1.

This refined context is used to inject failures of the different failure classes into the component automaton. A failure context is constructed by altering the context automata of the refined context of Step 2 such that the context automata model incoming failures of different failure classes. We create the set of failure contexts by constructing a failure context for each possible combination of failures that may be injected into the component. For each failure context, we build the reachable behavior, namely the failure zone graph.

In Step 3, we compare the zone graph, that represents the system behavior, with the set of failure zone graphs. From this, we construct the component’s TFPGs. Propagation times of failures are added to the TFPGs by analyzing the propagation times of paths in the zone graphs.

The TFPG of a component instance configuration is built by connecting the TFPGs of all component instances by the connectors. We assign the connectors’ propagation intervals to the newly created edges.

In the remainder, we assume that the whole behavior of a software component is specified by one timed automaton only. Each location of a timed automaton that has outgoing edges must have an invariant assigned that limits the time the timed automaton is allowed to stay in this location. If no invariant was specified, it would be possible to stay in a location infinitely long. This would result in an infinitely long propagation time which is not defined for TFPGs.

We now explain the steps in more detail.

A. Step 1 – Computing the Reachable Behavior of the Software Component

First, we compute the zone graph of the component type’s behavior. This behavior is specified by a CCN that consists of the component automaton A and an initial context. The initial context contains one timed automaton with an output action for each input action of A and one timed automaton with an input action for each output action of A. Each timed automaton of the initial context consists of one location and a self-transition that carries either an input or output action. These timed automata do not have any time constraints. As a consequence, the transitions may fire at any time. This allows to compute the maximum reachable behavior of the component automaton.

![Fig. 5. Initial Context Automata of the Timed Automaton of Figure 2](image)

The initial context for the timed automaton of Fig. 2 is shown in Fig. 5. Figures 5(a) and 5(b) show the timed automata that send $p_1.i_1!$ and $p_2.i_2!$ which are received as inputs by the timed automaton of Fig. 2. Figures 5(c) and 5(d) show the timed automata that receive the actions $p_3.o_1?$ and $p_5.o_2?$ that are sent as output actions by the timed automaton of Fig. 2.

![Fig. 6. Zone Graph Resulting from the Reachability Analysis](image)

Figure 6 shows the zone graph for the CCN consisting of the timed automaton shown in Fig. 2 and the initial context of Fig. 5. The zone graph contains all three traces that correspond to the executions of the automaton. A trace is a path in a zone graph. In Fig. 6, one of the three traces is highlighted as Trace1. We will use this trace to illustrate the generation of TFPGs in Section IV.

In zone graphs, three kinds of transitions may occur that represent different actions of the CCN. First, the CCN may delay, i.e., no transition fires and time passes. Second, if the transition is marked with $\tau$, a single timed automaton in the network fires a transition without any further synchronization. Third, if two transitions of different timed automata exist that use the same action name while one is an output action (!) and the other one is an input action (?), the two transitions may fire synchronously. For synchronous firing, we ignore the $p$ and just consider the action name $a$ of $p.a$. This is possible as it does not matter at which port a action is sent or received for analyzing the reachable behavior.

B. Step 2 – Computing the Reachable Behavior with Injected Failures

We now use the zone graph from Step 1 to create the refined context of the component automaton. The refined context has one context automaton for each transition in the zone graph that contains an input action. This allows to inject incoming failures for each occurrence of an input of the component automaton separately. Additionally, we create one context automaton for each output action of the component automaton.

We create a context input automaton for each transition in the zone graph that carries an input action. The context input automaton consists of two locations and one transition that has the same action $a$ as an output action. The time guard of the transition limits the execution of the transition to the time interval defined by the minimum and maximum time needed from the initial state of the zone graph to the transition that carries action $a$. Each input automaton has its own clock. The
clock starts with zero at the same time that the component automaton starts. This model of the context allows inputs to only fire once. Cycles in the reachable behavior can not be taken into account. We therefore leave cycles in the reachable behavior for future work.

We only explain the computation of the firing times for one clock here. The computation for more than one clocks is left to future work. As mentioned above, the firing time of the transition in the considered path in the zone graph is the sum of the time needed to traverse each partition. The traversal time of each partition is computed by the difference between firing time of the last transition and the clock value at the arrival at the first state of this partition. In our case, the clock value for the first state is always zero, because the first state is always the first state after a clock reset. Therefore, we simply take the firing time of the last transition. Except from the last partition, we have to compute this value from the clock constraints of the corresponding transition in the component automaton. This is because, the firing time of a transition is always represented by the state in the zone graph at which the transition is directed. Since we partition at clock resets, this clock constraint is always equal to zero.

For Partition 1 of Fig. 8(a) we compute the traversal time as follows. The corresponding transition of $(s_7, s_8)$ in the zone graph of Fig. 6 is $(l_1, l_3)$ in the component automaton of Fig. 2. The firing time is specified by the time invariant of location $l_1$ which is $c_1 \leq 15$ and the time guard at transition $(l_1, l_3)$ which is $c_1 \geq 10$. The minimum firing time of this transition is thus 10 and the maximum firing time is 15. The traversal time of Partition 1 is therefore minimum 10 and maximum 15 time units. The traversal time of Partition 2 is computed analogously. It takes 5 to 10 time units to traverse it. Thus the minimum firing time of transition $(l_3, l_4)$ in the component automaton of Fig. 2 is $10 + 5 = 15$ and the maximum firing time is $15 + 10 = 25$. This firing times is specified by the time guard of the input automaton of Fig. 7(c).

In addition, we create a context output automaton for each transition in the zone graph that carries an output action. The automaton consists of two locations and one transition that has the same symbol as an input action. In contrast to a context input automaton, the transition does not carry a time guard because we do not restrict the time interval in which the output must be produced. Then, the output actions of the component automaton can be sent at any time, particularly outside their intended time intervals. This enables the occurrence of outgoing timing failures. An example of a context output automaton is shown in Fig. 7(d).

We now build the failure zone graphs, i.e., the zone graphs that represent the reachable behavior of the component automaton with injected failures. Therefore, we first build the set of failure contexts from the refined context of Step 2. The failure contexts are built by replacing context input automata from the refined context with failure context automata. Failure context automata are context input automata that are altered such that the output event of their transition injects a failure into the component automaton.

For each incoming failure of a failure class, we create the following failure context automata. For incoming value failures, we create one failure context automaton for each action that is defined in the alphabet of the component automaton. For incoming service failures, we create one empty failure context automaton that consists only of a location an no transitions. For incoming timing failures, we create one failure context automaton that sends the output action early and one that

![Fig. 7. Refined Context of Trace1](image)

Figure 7 shows an excerpt of the refined context of the zone graph of Fig. 6 – the context of Trace1. In contrast to the initial context as depicted in Fig. 5, the context automata now have two separate locations. For the context input automata in Fig. 7(a), 7(b), and 7(c), we specify time guards that restrict the respective input to occur in the time interval that is expected by the component automaton. In Fig. 7(a), e.g., the transition specifies the time guard $5 \leq c_2 \leq 10$ for $p_1.i_1$ because $p_1.i_1$ is expected in this time interval as shown in the zone graph of Fig. 6.

The clock constraints of the input automata represent the earliest and latest absolute firing time of the transition of these automata. The firing times correspond to the time that passes during the traversal from the initial state to the target state of the transition which is triggered by the input automaton. At first, the traces of the zone graph are partitioned. The partitions are created by deleting all transitions in the zone graph that correspond to transitions with clock resets in the component automaton. The partitioning simplifies the computation of the firing times: The firing time of a transition is the sum of the traversed time intervals needed to traverse each partition which is the difference between the firing time of the last transition and the firing time of the first transition of the partition.

![Fig. 8. Partitions of Trace1](image)

Figure 8 shows the two partitions which are created when computing the minimum and maximum time needed to traverse Trace1 from the initial state to transition $(s_9, s_{10})$ where $p_1.i_1$ is received for the second time. These times are needed to create the clock constraints for the input automaton of Fig. 7(c). The partitions are created by deleting transition $(s_8, s_9)$ which corresponds to transition $(l_1, l_3)$ in the component automaton.
sends it late. The usage and construction of these automata is explained below.

For reasons of space, we limit the illustration of each failure context to the context automata that are needed for Trace1. For the generation of TFPGs from the component automaton of Fig. 2, of course, all context automata and all traces of the zone graph are needed.

In the following, we define failure automata that specify incoming failures of the different failure classes to construct the set of failure contexts of a component automaton. The zone graphs for these failure contexts are shown in Step 3 in Sec. IV-C.

1) Value Failures: A value failure occurs if the other message is sent or received than was intended by the developer. We thus model incoming value failures by sending a different action instead of the expected one. Therefore, we replace the output action of the respective context automaton with another output action of the component automaton. The port prefix of the output action is kept as the new action is received by the same port.

Fig. 9. Failure Context for a Value Failure

Figure 9 shows an excerpt of the failure context that injects a value failure on the first occurrence of $p_1.i_1$ and of $p_2.i_2$ of Trace1. The actions $p_1.i_1$ and $p_2.i_2$ of the context automata of Fig. 7(a) and (b) have been exchanged. The resulting failure context automata are shown in Fig. 9(a) and (b). The action of the automaton of Fig. 9(a) has been replaced with $p_1.i_2$ which models the fact that the action $i_2$ is received at port $p_1$. Analogous for the timed automaton of Fig. 9(b) the action $p_2.i_2$ is replaced with $p_2.i_1$. The context automata of Figs. 9(c) and 9(d) are left unchanged.

2) Service Failures: A service failure occurs if a message that is expected is never received. We thus model an incoming service failure by an empty failure context automaton.

Fig. 10. Failure Context for a Service Failure

Figure 10 shows a failure context of Trace1 that provokes a service failure on $p_1.i_1$ of Trace1. The failure input automata in Figs. 10(a) and 10(c) are empty. They have replaced the context input automata of Figs. 7(a) and 7(c). As a consequence, the action $p_1.i_1$ not provided by the failure context. The context automata of Figs. 10(b) and 10(d) are left unchanged.

3) Timing Failures: A timing failure occurs when a message is not delivered within the proper time interval. We thus model an incoming timing failure by altering the time constraints of an input automaton such that an output action is sent outside its proper time interval. In this paper, we consider the timing failure classes early timing and late timing.

Fig. 11. Failure Context for a Late Timing Failure

Figure 11 shows an excerpt of the failure context of $A_1$ that provokes a late timing failure on the first occurrence of $p_1.i_3$ and of $p_2.i_2$ of Trace1. As a consequence $p_1.i_3$ is sent when the value of clock $c_2$ is greater than 10. But it would have been expected at a clock value less than or equal to 10.

C. Step 3 – Constructing the TFPGs

In this step, we use the zone graph of the component’s behavior constructed in Step 1 and the set of failure zone graphs constructed in Step 3 to build the component’s TFPGs. For this, we take the approach for fault tree construction of [17] and extend it by time and failure classes.

We identify failures by actions in the failure zone graphs that deviate from the actions of the zone graph of the component’s behavior. As described before, we distinguish between value failures, service failures, and timing failures. We now assume, the correct behavior would be to send action $a = p.m$ at time interval $i = v_1 \leq c_1 \leq v_2$. We found a failure if the failure zone graph contains no transition that sends $a$ and fires within $i$. If there is no transition that sends any action at port $p$, there is a service failure at $p$, because the component does not send an output at $p$. If there is a transition that fires within $i$ but sends another action at $p$, this is a value failure at $p$. If there is a transition the sends $a$ but does not fire within $i$, this is a timing failure.

We compare the behavior represented by a failure zone graph with the zone graph of the correct behavior and identify its failure class. To identify all outgoing failures, we have to compare each transition in the zone graph of the component’s behavior to each transition of all failure zone graphs.

For each failure class of a port we create a separate TFPG. For each path in the failure zone graph that has an outgoing failure, we create a TFPG that is a tree. The outgoing failure is the root. The incoming failures located on the path are represented by the leaves which are connected to an AND-node. The AND-node is connected to the outgoing failure.
Each edge in this TFPG corresponds to a path in the zone graph. The propagation time intervals at the edges of the TFPG are calculated as explained in Sec. IV-B.

In the remainder, we explain the construction of TFPGs from the failure zone graphs of the failure contexts of Figs. 9 to 11.

![Diagram](image)

**Fig. 12. Zone Graph for the Failure Context of Figure 9**

Figure 12 shows the zone graph that represents the reachable behavior of the timed automaton of Fig. 2 and the failure context that is partly illustrated in Fig. 9. The zone graph consists of one path that leads from the start state  $s_0$ to state $s_2$. It contains no trace that is equivalent to Trace1. The path of this zone graph traverses only one transition with an output action, namely $p_{3,0_2}$ in the time interval $i = 5 \leq c_1 < 15$. There exists no transition with the output $p_{3,0_1}$. But $p_{3,0_2}$ appears at the same time interval when $p_{3,0_1}$ appears in the zone graph of Fig. 6. As a result, the component automaton sends $p_{3,0_2}$ when $p_{3,0_1}$ is expected. We thus obtain an outgoing value failure at $p_3$, if there are value failures at $p_1$ and $p_2$.

![Diagram](image)

**Fig. 13. TFPG resulting from the failure zone graph of Fig. 12**

Figure 13 shows the TFPG that results from the analysis of the failure zone graph of Fig. 12. The root node is the failure $f_{c_1,p_3,v}$, because the behavior represented by the failure zone graph of Fig. 12 shows an outgoing value failure at port $p_3$. The leaves of the TFPG of Fig. 13 are the failures $f_{c_1,p_1,v}$ and $f_{c_1,p_2,v}$, because they cause $f_{c_1,p_3,v}$. The edges originating from $f_{c_1,p_1,v}$ and $f_{c_1,p_2,v}$ point to an AND-node, because both failures are located on the same path in the failure zone graph. Consequently, both have to occur in order to let $f_{c_1,p_3,v}$ occur.

![Diagram](image)

**Fig. 14. Zone Graph for the Failure Context of Figure 10**

Figure 14 shows the zone graph that represents the reachable behavior of the component automaton of Fig. 2 and the context of Fig. 10. This zone graph only contains the states $s_0$ and $s_1$. Both are connected by a transition that has a delay. There is no transition that has an output action. This means that no output action can be produced by the component automaton of Fig. 2 in the context of Fig. 10. We thus obtain an outgoing service failure, if there is an incoming service failure at port $p_1$.

![Diagram](image)

**Fig. 15. TFPG resulting from the failure zone graph of Fig. 14**

Figure 15 shows the TFPG that results from the failure zone graph of Fig. 14. The propagation time from incoming service failure to outgoing service failure is the same as the propagation time of the correct data.

![Diagram](image)

**Fig. 16. Zone Graph for the Failure Context of Figure 11**

Figure 16 shows the corresponding zone graph for the timed automaton of Fig. 2 and the failure context of Fig. 16. Again, the zone graph contains no path that is equivalent to Trace1 because $p_1,i_1$ is only received when the value of clock $c_1$ is greater than 10. Thus, the transition from $l_0$ to $l_1$ is not enabled. Instead, the timed automaton of Fig. 2 executes the path containing the locations $l_0$, $l_5$, and $l_6$. That causes the output $p_{3,0_1}$ to be sent in the interval $30 < c_1 < 40$ and, thus, the output is too late.

![Diagram](image)

**Fig. 17. TFPG resulting from the failure zone graph of Fig. 16 and the TFPG of Fig. 13**

Figure 17 shows the TFPG that results from the failure zone graph of Fig. 16 and the TFPG of Fig. 13. The incoming late timing failure $f_{c_1,p_1,l}$ has been appended to the TFPG of Fig. 13, because $f_{c_1,p_1,l}$ causes the same outgoing failure $f_{c_1,p_3,v}$ which is the root node of the TFPG of Fig. 13. If more
than one path leads to the same outgoing failure, the AND-nodes of the TFPGs of the path are connected to an OR-node. This OR-node is then connected to the outgoing failure.

To construct all TFPGs of a component type, the method described above is applied to all failure zone graphs that were created in Step 2.

V. RELATED WORK

Kuntz et al. [10] presented an approach to generate fault trees from counterexamples of a probabilistic model checker. The system is modeled as a continuous time Markov chain. However, model checkers for Markov chains aim at checking probabilities of system properties [11], e.g., the probability that a state is reached within a certain time. It is not possible to compute the minimum and maximum runtime of a trace as is needed for timed hazard analysis [14].

Liggesmeyer et al. [13] inject failures into their behavior models and also use a model checker to generate fault trees. We adapt this technique and extend it by also analyzing the propagation times of failures. Further, in contrast to [13], we classify failures by failure classes. This allows for a more precise specification of failure propagation between components.

VI. CONCLUSIONS AND FUTURE WORK

We have presented an approach for the automatic generation of timed failure propagation graphs (TFPGs) from timed automata. For this, we determine the reachable behavior of a software component. Then we inject erroneous inputs into the behavior model of the software component and compare the reachable outputs to those that were reachable from original behavior. We construct the TFPG from this information. Propagation times of failures are added to the TFPGs by analyzing the propagation times of paths in the zone graphs.

The automatic generation of TFPGs relieves the developer from the tedious and error prone task of constructing TFPGs manually. Particularly, the manual estimation of propagation times is very difficult. The generation allows to take into account the whole behavior of a component type. This behavior is verified and thus guaranteed to satisfy the system requirements. The automatic generation guarantees that nothing is overlooked when constructing the TFPG.

In future works, we plan to evaluate our approach in a case study on the RailCab². The vision of the RailCab project is a mechatronic rail system with autonomously driving rail vehicles. This system is currently under construction and a first version of its controlling software has been built [8].

Further, we want to investigate how we can divide components into subcomponents such that the timed automaton of each subcomponent only has a limited amount of transitions with input actions. In this way, we can reduce the complexity of the generation of TFPGs for each component.

ACKNOWLEDGMENTS

This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

We thank Markus von Detten and Steffen Priesterjahn for proofreading and useful comments.

REFERENCES