Please note that this document is not entirely up to date with respect to the conference submission. In particular, some of the notations have been changed.

University Twente,
Faculty EWI,
Computer Science Department,
Formal Methods and Tools Group

Compositional Analysis of Dynamic Fault Trees using Input/Output Interactive Markov Chains

Master's thesis of Pepijn Crouzen

Supervisors: Dr. Mariëlle Stoelinga
Dr. Hichem Boudali
Dr. ir. Arend Rensink

Abstract

Dynamic fault trees (DFT) are widely used to analyze the fault-tolerance of computer systems. The syntax and semantics of DFT, however, lack formal definitions which has lead to vagueness in the interpretation of DFT. Existing analysis techniques also suffer from the state-space explosion problem.

This thesis describes a compositional approach to formalizing the DFT syntax and semantics. The DFT semantics are formalized by separating a DFT into its elements and formalizing the behavior of each element using an input/output interactive Markov chain (IOIMC). IOIMC are a combination of continuous-time Markov chains (CTMC) and input/output automata and IOIMC models can be combined using parallel composition. The semantics of a DFT are now defined as the composition of the semantics of its elements.

This compositional approach to formalizing DFT semantics also leads to the compositional analysis of DFT using compositional aggregation. Compositional aggregation is a well known technique which combines composition and aggregation to combat the state-space explosion problem.

In this thesis we give new formal definitions for the syntax and semantics of the DFT formalism. The IOIMC formalism is also introduced. Compositional analysis for DFT is then described which is accomplished using a translation tool (used to find IOIMC representations of DFT elements) and the Tipp tool. Two case studies are given that show the applicability of the new compositional analysis.

Contents

Chapter 1

Introduction

Computer systems fail. No matter how well designed, carefully implemented and thoroughly tested there is always a chance that a computer system will fail. For simple applications an occasional failure does not cause too much trouble. You might lose your last changes or even the contents of an entire file system, but we can mitigate such problems by making regular back-ups. There are, however, applications that can never really afford to fail like systems that control airplanes, medical devices or even spacecraft.

One well known strategy to overcome this problem is to build fault-tolerant computer systems [19]. The essential idea is to build a system that continues to operate even if one or more of its components fail. Fault tolerance is almost always the result of using redundant components. An everyday example is a car that carries an extra, redundant wheel. When you get a flat tire the failed component is simply replaced by the spare and the car can keep operating. Of course fault tolerance does not mean fault immunity. If you get two flat tires and you only have one spare your car has officially failed. But using redundant components does greatly decrease the chance that a system will fail.

When designing a fault tolerant computer system it is of great interest to quantify how fault tolerant the system is. It is also very useful to be able to compare the fault tolerance of several different alternatives. Typical measures of interest include the mean time until a system fails or the probability that a system will fail in a certain time period. For example: a communication network may be required to have less than 1% down-time, the chance that the flight system of a spacecraft fails during its 10 year mission might have to be less than .01% or the most reliable setup for a parallel processor system may need to be found.

1.1 Existing techniques for Fault tolerance analysis

A widely used mathematical approach to fault-tolerance analysis is using continuous-time Markov chains (CTMC) [18] [29]. CTMC are a well-defined mathematical formalism that can be used to describe the probabilistic behavior of a system (usually describing its lifetime from fully operational to system failure). Standard solution techniques, such as forward Chapman-Kolmogorov differential equations, can then be used to analyze the CTMC further. Transforming a complex computer system into a CTMC and then analyzing it can be a very labor-intensive process. The problem is that there are usually a large number of events that

can happen at any time and a CTMC has to model the consequence of all these events in all its states. This is a well known problem in CTMC analysis called *state space explosion*. Variations on CTMC have been proposed to allow multiple CTMC to be combined into one [17]. In this way a large system can be analyzed by finding CTMC for its components and then combining these into one large CTMC.

Static fault trees (SFT) [32] [31] are used to model the way different events influence each other and thus showing when the system failure event occurs. There are very powerful solution techniques for static fault trees such as binary decision diagram analysis [10] [27] and the modelling of a system using static fault trees is relatively easy. Static fault trees, however, cannot be used to model dynamic behavior (such as dependencies between events or the use of spare components) and therefore have limited use.

An extension to static fault trees called dynamic fault trees (DFT) was introduced [11] to make fault trees more expressive. Although DFT models can be used to describe dynamic behavior they do have a number of drawbacks as we will see in the next section. A lot of research has been done to extend the fault tree formalism in different ways such as by introducing coverage models [12], parametric fault trees [5] and repairable events [26]. A detailed discussion of static and dynamic fault trees can be found in Section 2.

1.2 Motivation

In this section we discuss some of the problems encountered in DFT analysis. These problems are the motivation for our research. In Section 1.3 we will show in general terms how we have sought to solve these problems.

First of all the DFT formalism lacks a formal syntax and a formal semantics, although some research has been done in this area [9]. This lack of formalization can lead to misunderstandings in the interpretation of DFT, for instance it has long been unclear what exactly happens when two events occur at exactly the same time (see Section 4.6.5).

Secondly the current analysis techniques for DFT (see Section 2.4) are not very compositional (i.e. a large DFT often cannot be split into subtrees for the purpose of analysis). A DFT is analyzed by generating a CTMC that represents the probabilistic behavior of the DFT. This CTMC is then analyzed further using CTMC analysis techniques as described in Section 1.1. Because of the state space explosion problem this CTMC can become very large even for small DFT. In Section 2 we will elaborate on this problem.

1.3 Approach

We have formally defined a syntax and a semantics for DFT in this thesis. The semantics of DFT are defined in a compositional way using input/output interactive Markov chains (IOIMC), a variation on interactive Markov chains (IMC) [15]. This semantics allows the DFT to be analyzed in a compositional way which mitigates the state space explosion problem.

We have formally defined the syntax of DFT by describing a DFT as a directed acyclic graph (DAG). On this DAG a number of restrictions are imposed to ensure that the DFT are valid (see Section 4.1). This removes any vagueness about what constitutes a valid DFT. The

semantics of DFT are formalized by modelling the behavior of the different elements (i.e. gates and basic events) of the DFT formalism using input/output interactive Markov chains (IOIMC).

IOIMC are a variation on IMC [15] which are in turn a combination of CTMC and interactive processes [24] (see Section 3). IOIMC allow the modelling of probabilistic behavior while also allowing communication between different models using discrete actions. These discrete actions allow the IOIMC to be composed using *parallel composition*. Furthermore techniques such as *abstraction* and *aggregation* are available to combat the state space explosion problem. The IOIMC formalism is discussed in detail in Section 3.

The IOIMC model of a DFT is found by first modelling the elements of the DFT as IOIMC and then using *compositional aggregation* to combine this group of IOIMC into a single IOIMC which is then analyzed to find the desired fault tolerance measures. Without compositional aggregation a model of the behavior of the DFT could be generated through parallel composition in a single step. This model would however be very large because of the state space explosion problem. Of course the large model could be aggregated, but this aggregation would be very labor-intensive because of the size of the model. Compositional aggregation avoids this state space explosion by performing the composition incrementally (i.e. composing only two sub-models at a time) and aggregating at every step. This approach to DFT analysis called *compositional analysis* is explained in detail in Section 5.

1.4 Results

The formalization of both the syntax and semantics of DFT have removed the vagueness from the DFT formalism and it is now clearly defined what a dynamic fault tree should look like and what such a DFT means. The syntax given in this thesis also extends the possibilities of the DFT formalism somewhat by allowing the modelling of complex spares and dependencies (see Section 4.2).

In this thesis the semantics of DFT are defined in a compositional way. This allows the analysis of DFT by using compositional aggregation. This compositional analysis mitigates the state space explosion problem and also allows the modularization of DFT analysis (i.e. the IOIMC model of a DFT can be reused in the analysis of other DFT). The compositional approach to defining DFT semantics also allows the DFT formalism to be easily extended. New DFT elements can simply be added to the formalism with their own IOIMC model describing their behavior. Such an extension would not influence the semantics of other DFT elements.

To implement the compositional analysis we have developed the DFT2Tipp tool to translate a DFT into IOIMC models. These IOIMC models are then composed and analyzed using the Tipp tool [16]. The tool support for this thesis is described in detail in Section 7.

The hypothetical cascaded priority-AND system (HCPS) case study (see Section 6.1) shows that the compositional analysis of DFT can avoid the creation of very large CTMC as happens with traditional analysis techniques. The cardiac assist system (CAS) case study (see Section 6.2) shows, however, that in some cases traditional analysis techniques are more efficient than compositional analysis. The case study also shows that there are still a lot of possibilities to increase the efficiency of compositional analysis, for instance the aggregation of IOIMC can be improved.

There is still a lot of room for further research in this area. Some of the possible avenues

of research are described in Section 8.3.

1.5 Organization of the thesis

This thesis consists of 8 chapters:

- This first chapter is the introduction to the rest of the thesis which gives a quick overview of the research described in detail in the rest of the thesis.
- Chapter 2 describes the existing DFT formalism as well as the existing analysis techniques for DFT.
- Chapter 3 describes the IOIMC formalism, its background and its technical details.
- Chapter 4 describes the formal syntax and semantics of DFT.
- Chapter 5 explains how we analyze DFT using compositional analysis.
- In Chapter 7 the tool support for our research is described.
- Finally our conclusions and suggestions for future work are presented in Chapter 8.

Chapter 2

Dynamic Fault Trees

In this section we present the dynamic fault tree formalism. This formalism can be used to model the failure behavior of computer systems. First we will introduce static fault trees in Section 2.1, which do not take into account the order in which components of the system fail. Then we present the expansion of the formalism to dynamic fault trees in Section 2.2 which can be used to model more complex behavior. DFT models can be analyzed using the DIFtree methodology [23]. We will explain how SFT and DFT are solved using this methodology in Section 2.4.

2.1 Static Fault Trees

Since as early as 1961 [32] [31] fault trees have been used to visualize the way computer systems, or systems in general, fail. An overview of the different symbols (basic events and gates) used in fault trees can be seen in Figure 2.1. Basic events generally correspond to a small component of the system and gates often represent larger components that are made up of several smaller components. A fault tree has a tree structure with gates as nodes in the tree, basic events as leaves and a single root-event displayed at the top of the fault tree. This root-event models the entire system. By using AND, OR and K/M gates (see Figure 2.1) we can show how different component failures (the inputs of the gate) lead to the failure of a larger component (the output of the gate). By connecting the inputs and outputs of gates and basic events the behavior of a large system consisting of many components and subcomponents can be described.

Figure 2.2 shows an example of a fault tree for a road trip. The road trip fails if the car breaks down (either because we get two flat tires or the engine fails) and we can not call road services because our mobile phone isn't working. The event 'Tires failed' represents the possibility that we might get two flat tires, which is problematic since we have only one spare. This event is modelled using a voting or K/M gate. In this case we use a 2/5 gate which means that the event occurs when at least 2 of its 5 input events has occurred. The five input events are four input events, that each model the failure of one tire. The fifth input event models the failure of the spare tire.

In this simplified example we model the failure of the car with an OR-gate which has as its two inputs the 'Tires failed' event and the 'Engine' basic event. This means that the car fails if

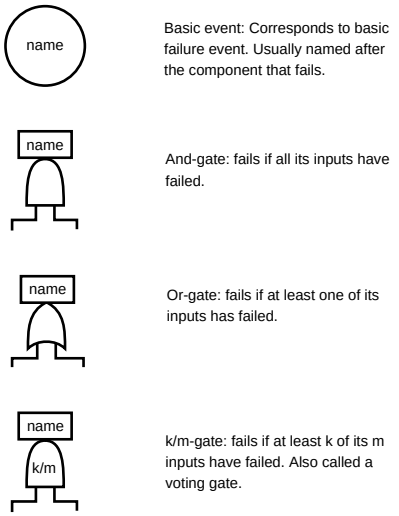

Figure 2.1: The elements used to create fault trees.

either the tires fail or the engine breaks down. Finally the entire road trip fails if both the car and the mobile phone fail. If only the car fails we can safely call road services and if only our mobile phone fails we can still finish our road trip without a problem. The top event of the SFT 'Road trip failed' is modelled as an AND-gate which fails when both of its inputs fail.

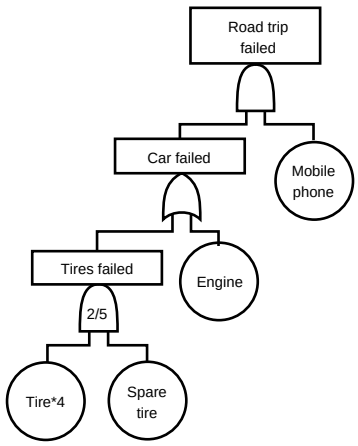

Figure 2.2: A fault tree for a road trip.

In the literature we see terms like 'events occurring', 'gates firing' and 'components failing'. These all mean exactly the same thing. In this thesis we will usually use the term *failing* when discussing a particular system and the term *firing* when talking about fault trees in general.

Static fault trees can be analyzed using binary decision diagrams [10] [27]. A binary decision diagram (BDD) is a compact representation of a logic formula. Since the top event of a static fault tree can be interpreted as a logic formula it can be transformed into a BDD. For our

example we can thus find a logic formula which describes exactly in what circumstances the road trip fails.

Every basic event fires after a certain delay. This delay is distributed probabilistically, ie. there is a probability x that a basic event fires within some time period t. In our road trip example we will assume that the delays of the basic events are all distributed exponentially. This means that the probability that the basic event occurs is related exponentially to the time period: $P(failure) = e^{-\lambda t}$, where λ is the *failure rate* of the event and t is the time period.

If we now assume that the mean time to failure (ie. the average lifetime) of a regular tire is 10000 hours while a spare tire has an average lifetime of 5000 hours. The engine is assumed to fail on average after 1000 hours and our very unreliable mobile phone fails once every 100 hours. Using BDD analysis we now find that for a 20 hour road trip, the probability that both the car and the mobile phone fail during the road trip is 0.0000394 or 0.00394 %. This measure is called the *unreliability* of the road trip, ie. the probability that a system fails within a certain time period. Although this result is quite reassuring for anyone who regularly has to make long trips we will see in the next subsection that this static fault tree does not really model the behavior of a road trip very accurately. With the introduction of *dynamic fault trees* in the next section we will be able to properly model the dynamic behavior of the road trip.

2.2 Dynamic Fault Trees

Static fault trees have been used frequently to model critical computer systems and determine their fault-tolerance [32] [31]. They do, however, have the shortcoming, among other things, that dynamic redundancy management cannot be modelled using fault trees. Figure 2.2, for instance, models the spare tire as just one of five components that may fail at any moment, but can a spare tire fail when it is in the back of your car? Certainly the probability of a tire failing dramatically increases when you start actually using it. To model such dynamic redundancy management we use dynamic fault trees [11]. Dynamic fault trees use the same elements as fault trees with a number of additional gates. These gates are shown in Figure 2.3.

Functional dependencies can be used to model the fact that one part of a system (the dependent event) is functionally dependent on another part of the same system (the trigger). The functional dependent gate denotes that when the trigger fails the dependent event will also fail. Functional dependencies are often used to model communication paths. For instance, when a cluster of computers is connected to a central mainframe via a bus the individual computers may be modelled as being functionally dependent on the bus. We can model this by making the events of the individual computers failing functionally dependent on the event of the bus failing. Now if the bus fails all the computers also become unavailable (essentially they fail) instantaneously.

Spare gates are used to model the use of spares. A spare gate has a primary input and one or more spare inputs, representing a primary component and spare components. Initially the spare components lie *dormant*. When the primary component fails the first available spare is activated. The failure of the first spare leads to the activation of the second available spare until there are no more spares available in which case the spare gate fails.

Priority-AND gates are used to model situations in which components must fail in a certain order to cause the failure of the PAND gate. When the inputs of the PAND gate have all failed

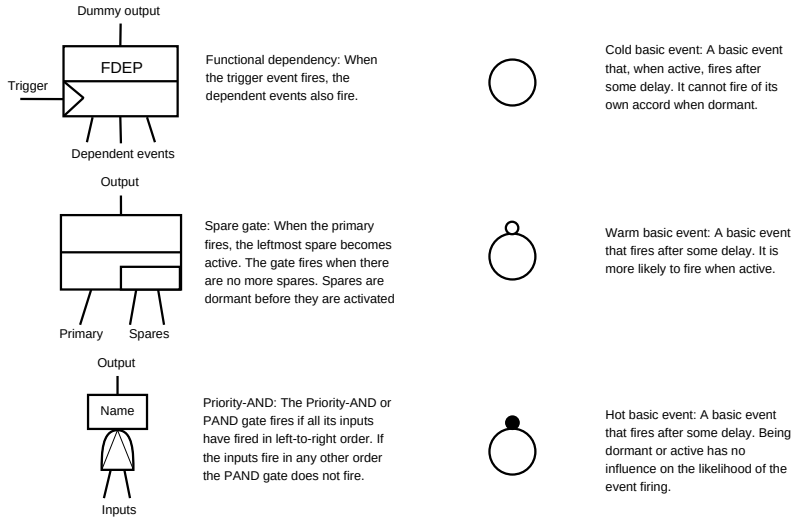

Figure 2.3: Dynamic fault tree gates.

in left-to-right order, the PAND gate also fails, but when one of its inputs fails out of order the PAND gate can never fail.

Basic events in dynamic fault trees come in three different temperatures: cold, warm or hot ones. This temperature refers to the behavior of the basic events when they are dormant. A basic event is dormant when it is part of a spare which has not yet been activated. Cold events cannot fire by themselves when dormant, warm events are less likely to fire when dormant and hot events fire equally quickly when dormant or active.

2.3 Example

Figure 2.4 shows a dynamic fault tree that models our road trip using spare gates to accurately show the use of the spare tire. Note that the spare tire is a so called *shared spare* in this DFT because it can be used to replace any of the four tires. The shared spare can obviously only be used to replace one tire. It then becomes unavailable for the other spare gates. We have chosen to model the tires and engine as cold basic events, which means we assume they cannot fail when dormant. The mobile phone, however, with its tendency to be dropped, lost or stolen is modelled as a warm spare, although in this system it is always active since it is not used as a spare. In Figure 2.4 the top-node is a priority-AND gate. This means that we only consider the road trip a failure if first our mobile phone fails and then our car. We do not consider the possibility that we get car trouble, call a repairman, continue our road trip and then get car trouble again. We could model this behavior by replacing the PAND gate with a spare gate or by extending the DFT formalism with the notion of repair (see [26] for an example of such an extension). For our DFT we could say that our road trip simply ends when we have to call a repairman. The next subsection explains how Dynamic Fault Trees can be analyzed by converting them into CTMC.

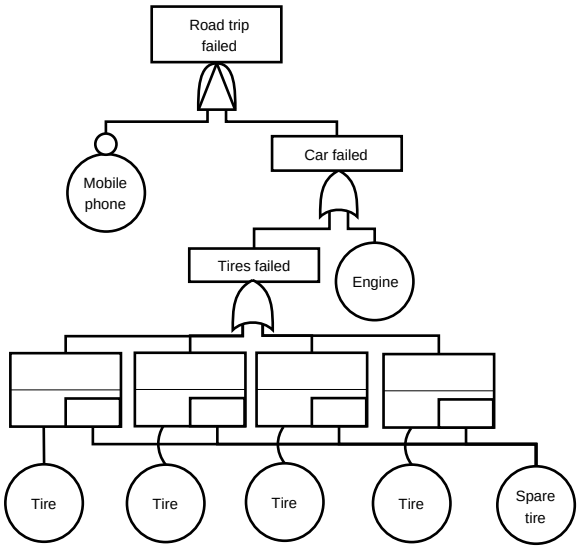

Figure 2.4: Dynamic fault tree for a road trip.

2.4 DIFTree

The main advantages of fault trees are their intuitiveness and readability. With the inception of dynamic fault trees this modelling technique has become even more expressive, being able to properly model dynamic redundancy management, sequential failures and dependencies. A drawback of dynamic fault trees compared to, for instance, CTMC is that they cannot be analyzed directly. In this section we will discuss at a widely used methodology to analyze DFT called DIFtree [14] [23]. Below we show how the unreliability of a system, modelled as a dynamic fault tree, can be determined using the DIFtree methodology. The unreliability of a system is the probability that it will fail within a certain time period, known as the *mission-time*.

1. The dynamic fault tree is split into independent sub-trees using a linear-time algorithm [13]. A sub-tree is independent when its gates share no inputs with gates outside the sub-tree. Dynamic sub-trees are not split into smaller sub-trees as modularization techniques do not provide an exact solution. This is explained in detail below.

2. For sub-trees that contain no other independent sub-trees the unreliability is calculated using 3 different techniques:

   (a) Static sub-trees, which consist only of static gates and basic events, are analyzed using Binary Decision Diagrams [10].

   (b) Dynamic sub-trees, which contain at least one dynamic gate, are analyzed using CTMC [11].

   (c) Sub-trees which cannot be analyzed by either of the above methods (often because they are too large) are analyzed using Monte Carlo simulation [7].

3. The solved sub-trees are replaced by basic events with a fixed failure probability in the DFT. This fixed failure probability is the unreliability of the sub-tree found in step 2.

4. Steps 2 and 3 are repeated until the DFT is completely replaced by one basic event. The unreliability of the entire DFT is now equal to the failure probability of this basic event.

The DIFtree methodology allows the time to failure for a basic event to be modelled using either fixed probabilities or probabilities with exponential, Weibull or Log normal distributions. In dynamic sub-trees fixed probabilities cannot be used as they are time-independent and the DFT gates are time-dependent (a basic event with a fixed failure probability has a constant unreliability, no matter what the mission-time is). The DIFtree methodology has been implemented in the *Galileo* tool [2], but it should be noted that Galileo does not support Monte Carlo simulation.

One of the problems of the DIFtree methodology is that large dynamic sub-trees cannot be solved accurately. Monte Carlo simulation gives inaccurate results for reasonable calculation times and using CTMC to solve these sub-trees results in huge CTMC that cannot realistically be analyzed. This shortcoming is made even more problematic by the fact that dynamic sub-trees cannot be split into smaller sub-trees. This is caused by the fact that sub-trees are solved to find their unreliability for a certain time period, in other words the probability that the top-level node of the sub-tree will fire within a time period. To analyze dynamic fault trees we however need to know the failure distribution of its subtrees, not just their unreliability.

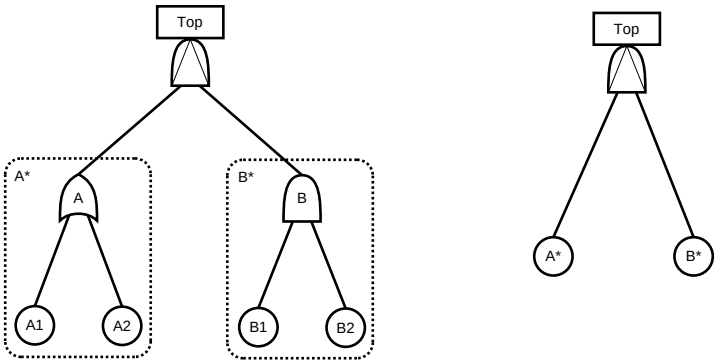

Figure 2.5: Example of a dynamic fault tree with two independent sub-trees (left) and the same DFT with the sub-trees replaced by basic events (right).

Figure 2.5 illustrates this problem. In step 2 of the DIFTree methodology the two subtrees A^* and B^* could be analyzed and replaced by basic events A^* and B^* with fixed probabilities as shown in the right part of Figure 2.5. Using this modified DFT we could easily find the probability that both events A^* and B^* fail within a certain mission-time, but because the top node of the DFT is a PAND gate we must calculate the probability that both A^* and B^* fail within a certain mission-time *and* A^* *fails before* B^*. Because we have replaced the two subtrees by basic events with fixed probabilities this calculation is no longer possible. The consequence is that the modularization technique of DIFTree cannot be applied to the DFT in Figure 2.5 and it must be solved by transforming it in its entirety to a CTMC.

Solving dynamic sub-trees using CTMC in the DIFtree methodology is performed as follows:

1. The start state of the CTMC is defined as the state in which no basic event has fired.

2. From any non-system-failure state X, transitions are added to (possibly new) states for every basic event Y that can fire in state X with as rate the failure rate of the failing component Y. The new state is defined as the state in which the basic events that had fired in state X, the basic event Y and its dependent events have fired.

3. Each newly added state is analyzed with regard to the DFT to see whether it is a system-failure state (i.e. a state in which the system, represented by the top node of the DFT, fails). If the DFT contains PAND-gates we can also identify states in which the DFT can not fail anymore. These states can also be disregarded in step 2.

4. Repeat steps 2 and 3 until no new states are added.

5. Calculate the system unreliability as the state-probability of the system-failure state of the created CTMC for a certain time period using forward Chapman-Kolmogorov differential equations.

A DFT with x basic events the corresponding CTMC can require in the order of $x! \cdot 2^x$ states using this method [6]. This number is reduced by functional dependencies and the fact that once the system has failed further component-failures are not considered. It can be further reduced by merging all the system-failure states into one system-failure state (the states from which the DFT can not fail anymore, because of a PAND gate, can also be merged). Even with these improvements the order of complexity for the transformation of DFT to CTMC remains high.

Chapter 3

Input/Output Interactive Markov Chains

In this chapter we discuss the formalism of input/output interactive Markov chains (IOIMC). This formalism is an integration of Input/Output Automata [21] and CTMC [18] [29]. IOIMC are closely related to Interactive Markov Chains [15] (IMC) which are an integration of interactive processes (IP) [24] and CTMC. We will first discuss why we have chosen the IOIMC formalism to model the semantics of DFT before formally defining IOIMC in Section 3.2. Finally we will define two equivalences on IOIMC in Sections 3.3 and 3.4.

3.1 Design choices

Figure 3.1 shows an example of an input/output interactive Markov chain. Circles denote states in the model and transitions are depicted as arrows. The starting state is identified by a black dot, in this case the starting state is $P1$. There are two different kinds of transitions in an IOIMC model: Markovian transitions, denoted by a small rectangle on the arrow and interactive transitions, denoted by a line on the arrow.

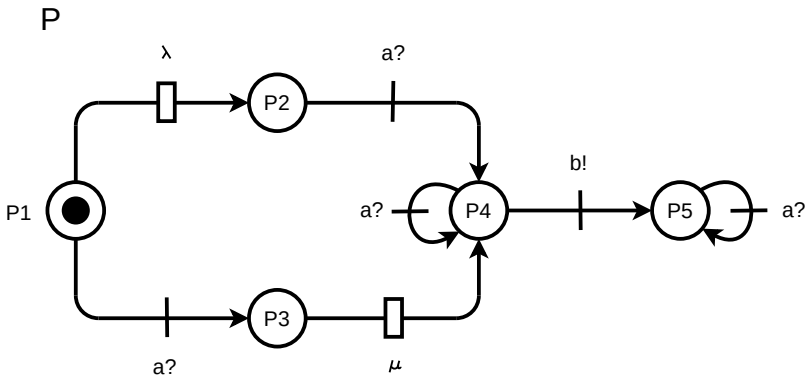

Figure 3.1: Example of an IOIMC: P.

In the following subsections we will explain exactly how the IOIMC P shown in Figure 3.1 behaves.

3.1.1 Probabilistic behavior

IOIMC P has a Markovian transition from state $P1$ to state $P2$. This transition has a *rate* of λ. Here, λ is a real, positive number which tells us something about when the transition from $P1$ to $P2$ will be taken. To be more specific, the rate λ tells us the probability that the transition is taken within a time period t:

$$\mathcal{P}(\text{IOIMC } P \text{ moves from state } P1 \text{ to } P2 \text{ within time period } t) = 1 - e^{-\lambda t}$$

We also say that the probability that P moves from $P1$ to $P2$ is *exponentially distributed* over the time-period t. Such an exponential distribution is *memoryless*, which means that the amount of time IOIMC P has already spent in state $P1$ has no influence on this distribution. And because exponential distributions are memoryless they adhere to the *Markov property* [18].

It is important to realize that a Markovian transition does not define *when* an IOIMC moves from a state to another but only gives us a distribution for this move. For instance, if $\lambda = \frac{1}{2}$ and we want to look at IOIMC P after 3 time-units we see that the probability that P has moved from $P1$ to $P2$ is $1 - e^{-\frac{1}{2}\cdot 3} = 0.78$ or 78%. Note that, if at $t = 3$ we know that P is still in state $P1$ then the probability that it moves to $P2$ within 3 time-units is once again 0.78 because of the memoryless property of exponential distributions.

An IOIMC with only Markovian transitions can be interpreted as a CTMC. The Markovian transitions in an IOIMC and the transitions in a CTMC behave in exactly the same way.

In this thesis we will use the Markovian transitions of IOIMC to model the firing of basic events in a DFT. These basic events in turn model the failing of components of a system. The rate λ of a Markovian transition is then equal to the inverse of the mean-time-to-failure of the component. This approach to modelling the failure of components using IOIMC assumes that the time-to-failure of a component is distributed exponentially. This modelling choice is discussed in more detail in Subsection 4.6.3.

3.1.2 Interactive behavior

In Figure 3.1 we can see that there is an interactive transition from state $P1$ to state $P3$ labelled $a?$. This denotes that the move from $P1$ to $P3$ is an *input action* named a. This means that if some other IOIMC performs an *output action* named a while IOIMC P is in state $P1$ then P will move to state $P3$ immediately. It is important to note that every state of IOIMC P has an outgoing input-action named a. This means that P is always ready to respond to an output-action a, even if this does not result in a state-change (when P is in state $P4$ or $P5$). We say that IOIMC P is *input-enabled* with respect to action a, because P is able to respond to action a in every state. Input actions are delayable. They must wait until another IOIMC performs the corresponding output-action.

A different kind of interactive transition goes from state $P4$ to state $P5$. This transition is labelled $b!$ and is an output action. When IOIMC P performs this output action all IOIMC which have b as an input action must perform these input actions. Output actions are immediate. This means that when IOIMC P moves to state $P4$ no *time* passes before it moves to state $P5$. It is however possible that another interactive transition is taken immediately. Specifically, if two or more different output actions are possible in a state, the choice between the transitions is

non-deterministic. One of the transitions is taken immediately, but it is not known how this choice is made.

Besides input and output actions there are also internal actions (which are not featured in the example IOIMC *P*). Internal actions do not influence other IOIMC and are not influenced by other IOIMC. Internal actions are immediate. For internal actions we find the same non-determinism as for output actions.

An IOIMC with only interactive transitions is isomorphic to an input/output automaton. As explained earlier IOIMC are closely related to IMC. The difference is that in IMC interactive transitions are not separated into input, output and internal actions. An IMC with only interactive transitions is then isomorphic to an IP [15].

In this thesis we use the interactive transitions of IOIMC to model communication between different gates and basic events in a DFT. When a basic event fires, for instance, it then signals its firing using an output action. Gates that have to respond to the firing of this basic event (because it is one of their inputs) then have a corresponding input action. This modelling choice is discussed in more detail in Subsection 4.6.1. Interactive transitions are also used to activate spares in a DFT (for a detailed explanation of this modelling choice see Subsection 4.6.2).

3.1.3 Compositional aggregation

The reason it is very interesting to combine Markovian and interactive transitions is that interactive transitions enable the construction of large IOIMC by composition of several smaller IOIMC. The subject at hand - the analysis of dynamic fault trees - is a good example. Instead of transforming the entire DFT into one large CTMC (see Section 2.4) we would like to transform the components of the DFT first and then create the total IOIMC by combining the smaller ones. The IOIMC formalism is one such approach to combining Markovian and interactive transitions, which is similar to the IMC formalism. A discussion of different approaches to combining Markovian and interactive transitions in one formalism can be found in [15].

An IOIMC can also be transformed into a smaller *aggregated* IOIMC that is equivalent with the original IOIMC (note that there are several different types of equivalences). This 'equivalence preserving state space aggregation' can very effectively reduce the resources necessary to create a model of some real-life system [17]. The technique of *compositional aggregation* consists of composing a large model out of smaller ones and aggregating sub-models after each compositional step (see Figure 3.2). Combining the formalisms of CTMC and input/output automata allows us to use compositional aggregation to analyze DFT: basic events are modelled as very simple IOIMC with Markovian transitions to model the delay before firing and gates are modelled as IOIMC that respond to signals from their inputs.

3.2 Input/Output Interactive Markov Chains

In this section we give the formal definition of an IOIMC as well as a number of additional definitions. We have arrived at the definition of IOIMC by applying the input/output notion to Interactive Markov chains following the application of this notion to IP in [21]. First we define *action signatures*.

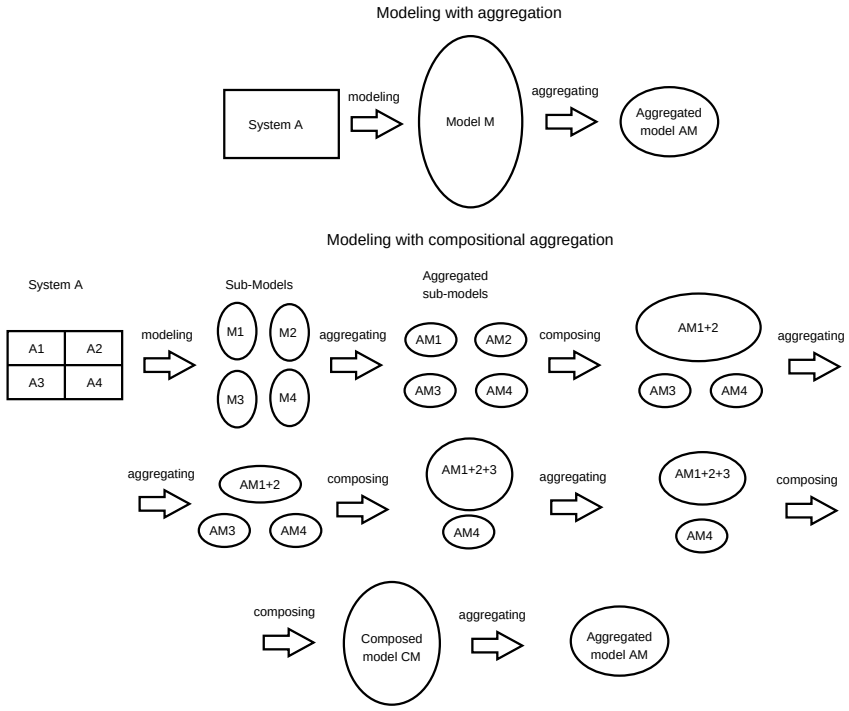

Figure 3.2: Example of compositional aggregation. Compositional aggregation avoids the construction and aggregation in one step of the large model M which would require a lot of resources.

Definition 3.1 *An action signature Sig is a partition of a set Act*(*Sig*) *of actions into three disjoint sets in*(*Sig*)*, out*(*Sig*) *and int*(*Sig*) *of input, output and internal actions, respectively.*

We also define the set of external actions of an action signature: $ext(Sig) = in(Sig) \cup out(Sig)$. We now use action signatures to define input/output interactive Markov chains.

Definition 3.2 *An input/output interactive Markov chain is a quintuple* $(S, Sig, \dashrightarrow, \dashrightarrow\!\!\!\!\!\square, P)$ *where*

- S *is a nonempty set of states,*
- Sig *is an action signature,*
- $\dashrightarrow \subset S \times Act(Sig) \times S$ *is a set of interactive transitions,*
- $\dashrightarrow\!\!\!\!\!\square \subset S \times \mathbb{R}^+ \times S$ *is a set of Markovian transitions,*
- $P \in S$ *is the initial state, and*
- *The IOIMC is input-enabled:* $\forall s \in S, a \in in(Sig) \cdot (\exists s' \in S \cdot s \overset{a}{\dashrightarrow} s')$

Note that any IOIMC can be considered as an IMC (with $Act = Act(Sig)$) and any continuous-time Markov chain can be considered as an IOIMC (with $Act(Sig) = \emptyset$).

Formally the set of actions for an IOIMC P with action signature Sig_P is $Act(Sig_P)$, but we will use $Act(P)$ as shorthand for the actions of P. $in(P)$, $out(P)$, $int(P)$ and $ext(P)$ can be used as shorthand for the input, output, internal and external actions of P, respectively. From now on we always suffix input-actions with a question-mark ($a?$), output-actions with an exclamation-mark ($a!$) and internal actions with a semi-colon ($a;$). Using this notation input-action $a?$ matches output-action $a!$. Note that the actual identity of action $a?$ is still a and not $a?$. So actions $a?$ and $a!$ are actually the same action in a different role. We can use this notation because an action can not have more than one role in an IOIMC (see Definition 3.1). An example of an IOIMC is shown in Figure 3.1.

To enable us to use compositional aggregation (see Subsection 3.1.3) we need a method to combine two IOIMC into one IOIMC. This method is called *parallel composition* and the idea is to look at the behavior of two IOIMC operating in parallel. Only *compatible* IOIMC can be composed. Two IOIMC are compatible if they do not share output-actions and if their internal actions are unique. When operating in parallel the IOIMC are *synchronized* on matching input and output actions *or* identical input actions in both IOIMC. Markovian transitions are simply interleaved.

Definition 3.3 *Let* $P = (S_P, Sig_P, \dashrightarrow_P, \dashrightarrow\!\!\!\!\boxdot_P, P)$ *and* $Q = (S_Q, Sig_Q, \dashrightarrow_Q, \dashrightarrow\!\!\!\!\boxdot_Q, Q)$ *be two IOIMC, with* $out(P) \cap out(Q) = \emptyset$, $int(P) \cap act(Q) = \emptyset$ *and* $int(Q) \cap act(P) = \emptyset$. *Parallel composition of* P *and* Q *is an IOIMC* $(S, Sig, \dashrightarrow, \dashrightarrow\!\!\!\!\boxdot, P\|Q)$, *where*

- $S := \{P'\|Q' \mid P' \in S_P \wedge Q' \in S_Q\}$,
- $out(Sig) := out(Sig_P) \cup out(Sig_Q)$,
- $in(Sig) := in(Sig_P) \cup in(Sig_Q) - out(Sig)$,
- $int(Sig) := int(Sig_P) \cup int(Sig_Q)$,
- $\dashrightarrow$ *is the least relation satisfying the last five rules in table 3.1, and*
- $\dashrightarrow\!\!\!\!\boxdot$ *is the least relation satisfying the first two rules in table 3.1.*

Note that $Act(Sig) = Act(Sig_P) \cup Act(Sig_Q)$.

Figure 3.3 shows an example of parallel composition. IOIMC P and Q both have three states. Their parallel composition $P\|Q$ could, at most, have nine states (three times three), but we can see that it only has six, because three of its possible states are unreachable.

From Definition 3.3 we can see that when parallel composing two IOIMC, output actions will always be immediate. This is caused by the fact that output actions are always synchronized with input actions (otherwise the IOIMC would not be compatible) and since IOIMC are input-enabled this means that if an output action is available in a state of one of the IOIMC it will also be available in the composite state.

We use *abstraction* to make actions in an IOIMC internal. This is useful when an action is no longer needed to communicate with other IOIMC. When an action is abstracted (or *hidden*) in an IOIMC its role is simply changed to internal.

1	$\dfrac{P \overset{\lambda}{\dashrightarrow} P'}{P\|\|Q \overset{\lambda}{\dashrightarrow} P'\|\|Q}$	
2	$\dfrac{Q \overset{\lambda}{\dashrightarrow} Q'}{P\|\|Q \overset{\lambda}{\dashrightarrow} P\|\|Q'}$	
3	$\dfrac{P \overset{a}{\longrightarrow} P'}{P\|\|Q \overset{a}{\longrightarrow} P'\|\|Q}$	$a \in act(P) \wedge a \notin act(Q)$
4	$\dfrac{Q \overset{a}{\longrightarrow} Q'}{P\|\|Q \overset{a}{\longrightarrow} P\|\|Q'}$	$a \notin act(P) \wedge a \in act(Q)$
5	$\dfrac{P \overset{a?}{\longrightarrow} P' \quad Q \overset{a?}{\longrightarrow} Q'}{P\|\|Q \overset{a?}{\longrightarrow} P'\|\|Q'}$	$a \in in(P) \wedge a \in in(Q)$
6	$\dfrac{P \overset{a?}{\longrightarrow} P' \quad Q \overset{a!}{\longrightarrow} Q'}{P\|\|Q \overset{a!}{\longrightarrow} P'\|\|Q'}$	$a \in in(P) \wedge a \in out(Q)$
7	$\dfrac{P \overset{a!}{\longrightarrow} P' \quad Q \overset{a?}{\longrightarrow} Q'}{P\|\|Q \overset{a!}{\longrightarrow} P'\|\|Q'}$	$a \in out(P) \wedge a \in in(Q)$

Table 3.1: Structural rules for parallel composition of IOIMC.

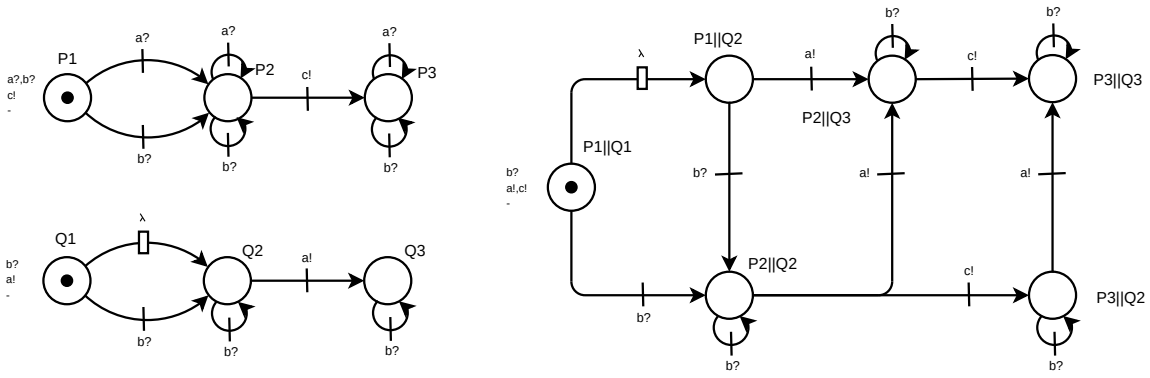

Figure 3.3: Example of parallel composition of IOIMC P and Q. The action signatures of the IOIMC are given besides their starting state.

Definition 3.4 *Let $P = (S, Sig, \dashrightarrow, \dashrightarrow, P)$ be an IOIMC. Abstraction of actions $a_1...a_n$ in P is an IOIMC $(S, Sig', \dashrightarrow, \dashrightarrow,$ **hide** $a_1...a_n$ **in** $P)$, where*

- $in(Sig') = in(Sig) - \{a_1...a_n\}$,
- $out(Sig') = out(Sig) - \{a_1...a_n\}$, *and*
- $int(Sig') = int(Sig') \cup \{a_1...a_n\}$.

We will model a DFT as a set of *communicating* IOIMC. We define such a set as a *community*, based on the notion of *compatibility* in [21].

Definition 3.5 *A community C of Input/Output Interactive Markov Chains is a set of IOIMC in which,*

- $\forall C_i, C_j \in C \wedge C_i \neq C_j : out(C_i) \cap out(C_j) = \emptyset$,
- $\forall C_i, C_j \in C \wedge C_i \neq C_j : int(C_i) \cap Act(C_j) = \emptyset$

So a set of IOIMC is a community if and only if no two members share an output action and an internal action of one member is never also an action of another member. An example of an IOIMC community (IOIMCC) is given in Figure 3.4.

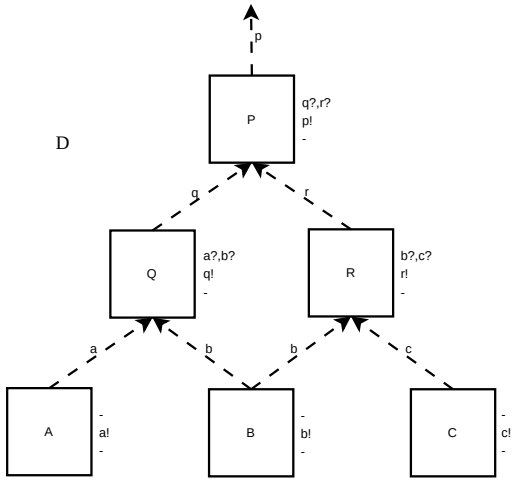

Figure 3.4: Example of an IOIMC community. To the right of each member its action signature is given. Dashed arrows denote communication between members.

So why use IOIMC communities instead of just IOIMC? The reason lies in the fact that we need to define exactly when certain actions are abstracted in an IOIMC. Let's consider the parallel composition of IOIMC Q and A in the IOIMC community shown in Figure 3.4. This parallel composition would result in an IOIMC $Q||A$ with input action $b?$ and output-actions $a!$ and $q!$. To determine which of these actions must be hidden we have to consider the other IOIMC in the community. For instance in IOIMC $Q||A$ the actions $b?$ and $q!$ should not be hidden, since IOIMC B and P use these actions to communicate with $Q||A$. Action $a!$ can, however, be hidden safely because no other IOIMC in the community is interested in this signal. This assessment is based on the assumption that the only signals used to communicate between the community and the 'outside world' are those input and output signals that do not have corresponding output and input signals within the IOIMC community. For the example in Figure 3.4 we find that the IOIMCC only communicates with the 'outside world' via output signal p.

All this means that, for IOIMC communities, we can get a well-defined definition of parallel composition followed by abstraction without having to specify (a) what actions we are synchronizing on and (b) what actions we are hiding, since both can be derived from the IOIMC community. We call this combination of parallel composition and hiding *abstracting composition* or combination. Before giving the definition of the abstracting composition in an IOIMC community we define action signatures for subsets of an IOIMC community.

Definition 3.6 *For a subset D of an IOIMC community C we define the following shorthand notations:*

- *The union of all the input actions of all the different IOIMC in D is given by: $all\_in(D) = \bigcup_{D_i}^{D_i \in D} in(D_i)$,*
- *The union of all the output actions of all the different IOIMC in D is given by: $all\_out(D) = \bigcup_{D_i}^{D_i \in D} out(D_i)$, and*
- *The union of all the internal actions of all the different IOIMC in D is given by: $all\_int(D) = \bigcup_{D_i}^{D_i \in D} int(D_i)$.*

We now define the action signature of D within the IOIMC community C as follows:

- *The input signature of D consists of all those input actions of elements of D that do not correspond to any output action in D:*

$$in(D) = all\_in(D) \setminus all\_out(D)$$

- *The output signature of D consists of all those output actions of elements of D that do not correspond **only** to input actions within D:*

$$out(D) = all\_out(D) \setminus (all\_in(D) \setminus all\_in(C \setminus D))$$

- *The rest of the actions of D are of course internal:*

$$int(D) = \bigcup_{D_i}^{D_i \in D} act(D_i) \setminus (in(D) \cup out(D))$$

Figure 3.5 shows a schematic of the action signature of a subset D of an IOIMCC C. We see three different types of output actions of D: output actions that do not correspond to any input action in C (1), output actions that correspond to a number of input actions in C but outside D (2) and output actions that correspond to a number of input actions both in- and outside D (3). There are also two different types of input actions: input actions that do not correspond to any input action in C (4) and input actions that correspond to an output action in C but outside D (5). Finally internal actions of D are either internal actions of some IOIMC in D (not shown in Figure 3.5) or output actions in D that correspond to a number of input actions in D but not to input actions in C (6).

Definition 3.7 *Let $C = \{C_1, \ldots, C_n\}$ be an IOIMC community. Let $D = \{D_1, \ldots, D_m\}$ be a set of IOIMC such that $D \subseteq C$. The abstracting composition of D in C is* **combine** D **in** C*,where:*

$$\textbf{combine } D \textbf{ in } C = (C - D) \cup (\textbf{hide } int(D) \textbf{ in } D_1 || \ldots || D_m)$$

Note that the action signature of **hide** $int(D)$ **in** $D_1 || \ldots || D_m$ is the same as the action signature of D.

Two examples of abstracting composition are given in Figures 3.6 and 3.7. IOIMC community $\mathcal{D}'$ is the abstracting composition of members A and Q of IOIMCC $\mathcal{D}$ in Figure 3.4. $\mathcal{D}''$ is the abstracting composition of members P and B in $\mathcal{D}$.

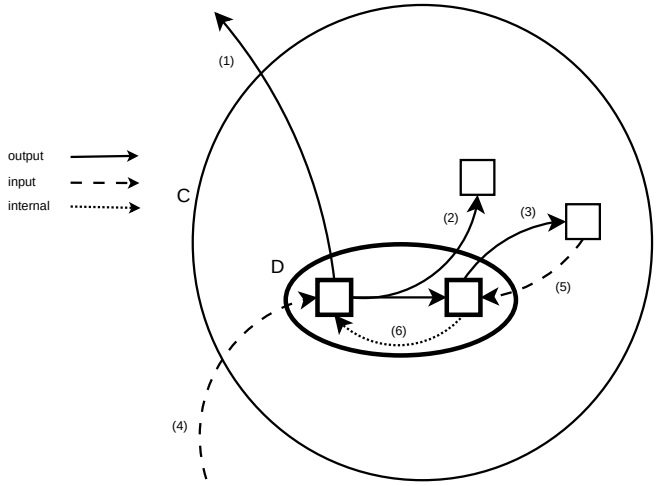

Figure 3.5: Schematic of the action signature of a subset of an IOIMCC

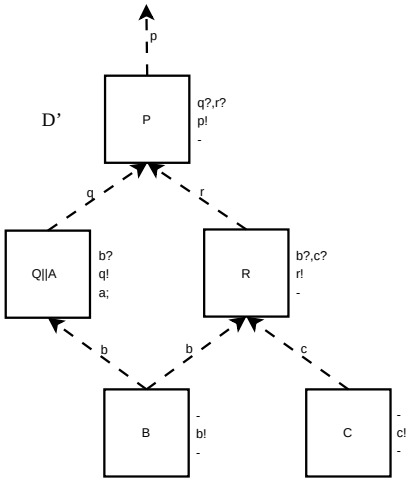

Figure 3.6: Example of abstracting composition

3.3 Strong bisimulation

Meaningful equivalences for IOIMC should be based on equivalences of both interactive processes [15, section 2.2] as well as continuous-time Markov chains [15, section 3.5]. In short, equivalences for interactive processes are based on the idea that equivalent states should have outgoing transitions with the same labels to equivalent states (so if a state P has an outgoing transition labelled a to a state S than any state equivalent to P should have an a-transition to a state equivalent to S). Equivalences of CTMC are defined similarly, taking into account the mathematical principles of exponential distributions.

Additionally, equivalences for IOIMC should address the way interactive and Markovian transitions behave with respect to each other. Consider, for instance the IOIMC shown in Figure 3.8 In its starting state this IOIMC can perform both an output transition or a Markovian

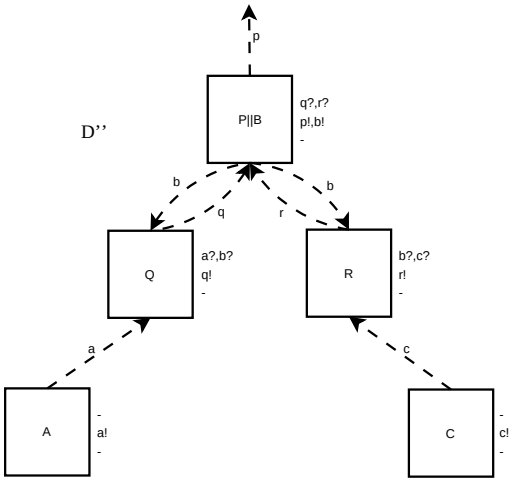

Figure 3.7: Example of abstracting composition

transition. The probability that the Markovian transition is taken immediately is zero. An output transition can be taken immediately since no outside influence can prevent or delay it (in contrast with input actions which can be delayed by the environment). Even if we parallel compose this IOIMC with another IOIMC the output transition cannot be delayed, since IOIMC are input-enabled. We assume that an IOIMC that may perform an output or an internal transition is not allowed to let time pass and will perform the output or internal transition immediately. This assumption is called the *maximal progress assumption* [15, section 4.2]. To define equivalences we must differentiate between stable and unstable states. Stable states,

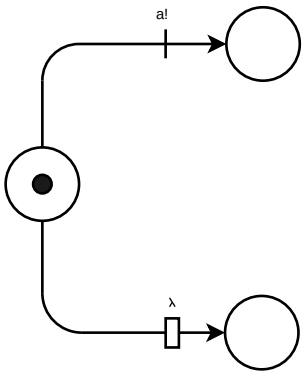

Figure 3.8: Example of an IMC.

denoted **stable**(P), are states that have no outgoing internal or output actions. Unstable states, denoted **unstable**(P), do have outgoing internal or output actions. The reason these states are called unstable is that, because of the maximal progress assumption, an IOIMC will always leave an unstable state immediately via one of its internal or output transitions.

To deal with Markovian transitions we define a *cumulative rate* function γ_{M}, which gives the sum of all rates of Markovian transitions from a state to a set of other states. $\gamma_{\text{M}}(R, C) =$

$\sum[\lambda | R \xrightarrow{\lambda} R' \wedge R' \in C]$. It is easy to see that $\gamma_M(R, C)$ equals the rate of the delay for the IMC moving from state R to a state in C, because the sum of rates equals the rate of the minimum delay for the relevant transitions. In the follow S^{all} stands for the superset of all appearing states and *Act* denotes all appearing actions.

We will now give the definition of strong bisimulation for IOIMC. Two IOIMC states are said to be strongly bisimilar if they have matching outgoing transitions and the same action signature. The action signature of a state is the same as the action signature of the IOIMC this state belongs to.

Definition 3.8 (Strong bisimulation) *Let $P = (S, Sig, \rightarrow, \Rightarrow, P)$ be an IOIMC. Let R be an equivalence relation on S. Then R is a* strong bisimulation *iff for all $(s,t) \in R$, $a \in Act(P)$*

1. *$s \xrightarrow{a} s'$ implies that there is a transition $t \xrightarrow{a} t'$ with $(s', t') \in R$.*
2. *s stable implies t stable and $\gamma_M(s, C) = \gamma_M(t, C)$, for all equivalence classes $C \in S/R$*

The states s and t in P are strongly bisimilar*, notation $s \sim_P t$, if and only if there exists a strong bisimulation R with $(s,t) \in R$. Strong bisimilarity for an IOIMC P is defined as the union of all strong bisimulations on P:*

$$\sim_P = \bigcup \{R \mid R \text{ is a strong bisimulation on } P\}$$

We often omit the name of the IOIMC if it is clear from context.

Coinductive definitions such as that of $\sim$ have shown to be very useful for other equivalence relations. We do have to show however that $\sim$ is indeed a strong bisimulation and also the largest one.

Theorem 3.1 *For relation $\sim$ we find:*

1. *$\sim$ is the largest strong bisimulation.*

We do not give the proof here, but it follows the lines of the same proof for weak bisimilarity (see Theorem 3.3).

The equivalence relation $\sim$ divides the set of all IOIMC states S^{all} into equivalence classes. An equivalence class is a subset of S^{all} which contains states that are all strongly bisimilar to each other. One useful aspect of such equivalence classes is that we can now 'loosen' our restrictions when analyzing IOIMC. This means that when talking about IOIMC being the same we could 'loosen' this concept from 'completely identical' to 'strongly bisimilar'. The idea here is that we don't care if two IOIMC are not completely identical as long as they are strongly bisimilar. For instance instead of analyzing a large IOIMC, which may be very costly, we could analyze a smaller, strongly bisimilar IOIMC. Let's say we want to compose a large IOIMC X with another IOIMC Z and these IOIMC contain n and m states respectively. The resulting IOIMC $X||Z$ can have as many as $m \cdot n$ states so it is very worthwhile to calculate this composition using a smaller IOIMC Y bisimilar to IOIMC X. Of course we must ensure that the resulting IOIMC $Y||Z$ is not suddenly different (with regard to strong bisimulation!) from $X||Z$.

Theorem 3.2 *Strong bisimilarity is substitutive with parallel composition and hiding.*

$$\begin{array}{lll} P_1 \sim P_2 & \textit{implies} & P_1 \| P_3 \sim P_2 \| P_3 \\ P_1 \sim P_2 & \textit{implies} & P_3 \| P_1 \sim P_3 \| P_2 \\ P_1 \sim P_2 & \textit{implies} & \textbf{hide } a_1, ..., a_n \textbf{ in } P_1 \sim \textbf{hide } a_1, ..., a_n \textbf{ in } P_2 \end{array}$$

We do not give the proofs here, but they follows the lines of the same proofs for weak bisimilarity (see Theorem 3.4).

3.4 Weak bisimulation

Weak bisimulation for input/output interactive Markov chains is defined along the same lines of weak bisimulation for IMC [15, section 4.4]. Like in strong bisimulation outgoing input and output transitions have to match in weakly bisimilar states, but now these transitions may be preceded and followed by a number of internal transitions. Markovian transitions can also be preceded or followed by a number of internal transitions.

Weak bisimulation equivalences identify states with the same visible behavior while preserving performance measures (e.g. reliability). Basically, two IOIMC states s, t are weakly bisimilar if whatever steps can be taken from s, can also be taken from t, and the target states are again bisimulation equivalent. Weak bisimulation abstracts from internal computation, i.e. one step in state s may correspond to the same step in state t preceded and followed by a number of internal transitions. Internal computation is formalized by the weak transition relation $\Longrightarrow$.

Definition 3.9 *Let $P = (S, Sig, \dashrightarrow, \dashrightarrow\!\!\!\!\square, P)$ be an IOIMC. We define the internal transitions relation $\dashrightarrow^{int}$ as the relation $\{(s,t) \mid (s,a,t) \in \dashrightarrow \wedge\ a \in in(P)\}$. We denote by $\Longrightarrow$ the transitive, reflexive closure of $\dashrightarrow^{int}$, i.e. we have $s \Longrightarrow s'$ if and only if s' can be reached from s via a sequence of internal transitions. For input and output actions a we write $s \overset{a}{\Longrightarrow} s'$ if there are states t, t' such that $s \Longrightarrow t \overset{a}{\dashrightarrow} t' \Longrightarrow s'$. For an internal action a $s \overset{a}{\Longrightarrow} s'$ simply means that $s \Longrightarrow s'$.*

Just like strong bisimulation, weak bisimulation abstracts from individual Markovian transitions and looks instead at cumulative rates, again respecting the maximal progress assumption. One of the differences between strong and weak bisimulation is that in weak bisimulation Markovian transitions can be preceded and followed by internal steps. To define the restriction on cumulative rates for weak bisimulation we must introduced the internal backward closure.

Definition 3.10 *[15] The* internal backward closure C^{int} *of a set of states C is defined as the set of states that may internally move to a state in C, i.e. $C^{int} = \{s' \mid \exists s \in C \cdot s' \Longrightarrow s\}$.*

Weak bisimulation also disregards *Markovian self-loops* [8]. This means that Markovian transitions are considered to be unobservable and Markovian transitions to equivalent states do not have to be simulated. The reason the restriction on Markovian transitions is weakened is

that, because Markovian transitions are memoryless, moving to an equivalent state does not affect subsequent behavior. In other words, if we have two weakly bisimilar states A and B the behavior of the IOIMC is not affected by the presence or absence of a Markovian transition from A to B.

Definition 3.11 (Weak bisimulation) *Let $P = (S, Sig, \dashrightarrow, \dashrightarrow\!\!\!\!\square, P)$ be an IOIMC. Let R be an equivalence relation on S. Then R is a* weak bisimulation *iff for all $(s,t) \in R$, $a \in Act(P)$*

1. *$s \stackrel{a}{\Longrightarrow} s'$ implies that there is a weak transition $t \stackrel{a}{\Longrightarrow} t'$ with $(s',t') \in R$.*
2. *$s \Longrightarrow s'$ and s' stable imply that there is a t' such that $t \Longrightarrow t'$ and t' stable and $\gamma_M(s', C^{int}) = \gamma_M(t', C^{int})$, for all equivalence classes $C \in (S/R) \setminus [s']_R$*

The states s and t in P are weakly bisimilar*, notation $s \approx_P t$, if and only if there exists a weak bisimulation R with $(s,t) \in R$. Weak bisimilarity for an IOIMC P is defined as the union of all weak bisimulations on P: $\approx_P = \bigcup\{R \mid R \text{ is a weak bisimulation on } P\}$. We often omit the name of the IOIMC if it is clear from context.*

It is important to note that weak bisimilarity really only differentiates on the (future) *output* behavior of a state. The fundamental difference between two states under weak bisimulation is always that one can perform some output action and the other cannot, even if this output action moves to a weakly bisimilar state. Since IOIMC are input-enabled we do not see such a difference for input actions and an internal action to a weakly bisimilar state also does not need to be simulated. Finally, Markovian transitions to weakly bisimilar states also do not need to be simulates since the second clause of weak bisimulation does not apply to weakly bisimilar states. This means that when two states are not weakly bisimilar because of an input, internal or Markovian transition, this difference is always instigated by another difference in the target states of the transitions. For an IOIMC has that no output transitions we find that all of its states must be weakly bisimilar (proof?).

Analogous to strong bisimulation we want to show that $\approx$ is the largest weak bisimulation and that it is substitutive with parallel composition and abstraction.

Theorem 3.3 *For an IOIMC P relation $\approx_P$ we find:*

1. *$\approx_P$ is the largest weak bisimulation on P.*

Theorem 3.4 *Weak bisimilarity is substitutive with parallel composition and hiding.*

$$
\begin{array}{lll}
P_1 \approx P_2 & \textit{implies} & P_1 \| P_3 \approx P_2 \| P_3 \\
P_1 \approx P_2 & \textit{implies} & P_3 \| P_1 \approx P_3 \| P_2 \\
P_1 \approx P_2 & \textit{implies} & \textbf{hide } a_1, ..., a_n \textbf{ in } P_1 \approx \textbf{hide } a_1, ..., a_n \textbf{ in } P_2
\end{array}
$$

The proofs for Theorems 3.3 and 3.4 can be found in Appendix A.

Chapter 4

DFT syntax and semantics

In this chapter we will formally define the syntax and semantics of the dynamic fault tree formalism. In Section 4.1 the syntax of DFT is formally defined. We then discuss how we have extended the DFT formalism to make it more modular. In Section 4.3 the concept of dormancy is discussed.

The second part of this chapter concerns the formalization of the semantics of DFT using IOIMC. We will start by explaining some notations we use in the IOIMC models of DFT elements in Section 4.4. Secondly examples of all the IOIMC models of DFT elements are given. Afterwards some design choices will be highlighted and explained before a formal translation of DFT elements into IOIMC models is given.

4.1 Formal DFT syntax

In this subsection we give a formal definition of the structure of dynamic fault trees. First we give a definition of the DFT elements and then the definition of a DFT itself.

Definition 4.1 *The set* ELEMENTS *is a set of tuples representing DFT elements. There are 7 different types of tuples:*

- OR *is a two-tuple consisting of the type of the OR-gate and a natural number of inputs:* OR $= (OR, n)$*, where* $n \in \mathbb{N} \wedge n \geq 1$.
- AND *is a two-tuple consisting of the type of the AND-gate and a natural number of inputs:* AND $= (AND, n)$*, where* $n \in \mathbb{N} \wedge n \geq 1$.
- VOTING *is a three-tuple consisting of the type of the VOTING-gate, a natural number of inputs and a natural number representing the threshold:* VOTING $= (VOTING, n, k)$*, where* $n, k \in \mathbb{N} \wedge 1 \leq k \leq n$.
- PAND *is a two-tuple consisting of the type of the PAND-gate and a natural number of inputs:* PAND $= (PAND, n)$*, where* $n \in \mathbb{N} \wedge n \geq 1$.
- SPARE *is a two-tuple consisting of the type of the spare gate and a natural number of inputs:* SPARE $= (SPARE, n)$*, where* $n \in \mathbb{N} \wedge n \geq 1$.

- FDEP *is a two-tuple consisting of the type of the FDEP-gate and a natural number of inputs:* FDEP $= (FDEP, n)$*, where* $n \in \mathbb{N} \wedge n \geq 2$*.*

- BE *is a four-tuple consisting of the type of the basic event, the natural number of inputs* 0*, the active delay rate and the passive delay rate of the basic event:* BE $= (BE, 0, \lambda, \mu)$*, where* $\lambda, \mu \in \mathbb{R} \wedge \lambda, \mu \geq 0$*. For cold basic events we find that* $\mu = 0$*, for warm basic events we find* $0 \leq \mu \leq \lambda$ *and for hot basic events* $\mu = \lambda$*.*

We also define the functions type and inputs which can be used to determine for any DFT element its type and the number of its inputs:

- $type$: ELEMENTS $\rightarrow \{OR, AND, VOTING, PAND, SPARE, FDEP, BE\}$ *is given by the first member of the DFT element tuple:* $type(E) =$ **first**(E)*.*

- $inputs$: ELEMENTS $\rightarrow \mathbb{N}$ *is given by the second member of the DFT element tuple:* $inputs(E) =$ **second**(E)*.*

A dynamic fault tree is described as a directed acyclic graph. The elements, basic events as well as complex gates, will be the vertices of this graph while the connections in the DFT will be the edges of the graphs. Edges will be directed from outputs (the 'top' of a DFT element) to inputs (the 'bottom' of a DFT gate).

Definition 4.2 *A dynamic fault tree is a quadruple* (V, in, l, T) *where:*

- V *is a set of vertices,*

- $in : V \rightarrow V^*$ *is a function that defines the connections between these vertices,* $w \in in(v)$ *means that there is an edge from vertex* w *to vertex* v*,*

- $l : V \rightarrow$ ELEMENTS *is a labelling function, and*

- $T \in V$ *is the top vertex which corresponds to the top element of the DFT.*

We define the function in' *that defines the non-dummy connections between the vertices (the outputs of FDEP-gates are dummy connections). We also define a set of edges* E *based on the function* in *and a function* in_{first} *which gives the first non-dummy input of a vertice:*

- $in' : V \rightarrow V^* = \{(v, W') \mid (v, W) \in in \wedge W' = \{w \mid w \in W \wedge type(l(w)) \neq FDEP\}\}$*,*

- $E : V \times V = \{(v, w) \mid w \in in(v)\}$*, and*

- $in_{first} : V \rightarrow V = \{(v, w) \mid in'(v) = W \wedge w$ *is the first element of* $W\}$*.*

The restrictions on DFT are given below, but first a set of independent vertices is defined using the concept of subtrees:

- *For any vertex* v *in the DFT we define the* ***subtree below*** v *as the vertex itself and the set of vertices from which the vertex is reachable:* $subtree : V \rightarrow \mathcal{P}V$*, where* ***subtree below*** $x = \{x\} \cup \{y \mid y \in V \wedge x$ *reachable from* y *in the directed graph* $(V, E)\}$*.*

- *A vertex v is **independent** when the vertices in the **subtree below** v share no inputs with gates outside the sub-tree. We define the set $indep$ as the subset of V that holds all the independent vertices: $indep \subseteq V$, where $x \in indep \Leftrightarrow \forall y \in$ **subtree below** $x \cdot (\exists z \in V \setminus$ **subtree below** $x \cdot y \in in(z) \rightarrow y = x)$.*
- *(V, E) forms a directed acyclic graph,*
- *All inputs of the elements must be connected: $\forall v \in V \cdot l(v) = G \Leftrightarrow inputs(G) = |in(v)|$,*
- *The top vertex may not have any connected outputs: $\nexists v \in V \cdot T \in in(v)$,*
- *The top vertex may not be an FDEP-gate: $type(l(T)) \neq FDEP$,*
- *Every vertex besides the top vertex must have a connected output: $\forall w \in V \setminus \{T\} \cdot (\exists v \in V \cdot w \in in(v))$,*
- *The first non-dummy input of a spare gate (the primary component) may not be an input to another spare gate (spare components may be shared between spare gates): $\forall w \in V \cdot (\exists v \in V \cdot w = in_{first}(v) \wedge type(l(v)) = SPARE) \rightarrow (\nexists x \in V \cdot x \neq v \wedge type(l(v)) = SPARE \wedge w \in in'(x))$, and*
- *Inputs of spare gates (both primary and spares) must be independent subtrees: $\forall w \in V \cdot (\exists v \in V \cdot w \in in'(v) \wedge type(l(v)) = SPARE) \rightarrow w \in indep$.*

4.2 Complex spares and dependencies

Looking at static fault trees (see Section 2.1) we can see that they are highly modular, i.e. we can take any static fault tree and use it as a module in another static fault trees (by attaching its top-node as an input to some gate). We want to achieve the same level of modularity for dynamic fault trees, because increased modularity extends the modelling capabilities of the DFT formalism. This does not seem like a big problem, but, for instance, the tool Galileo mentioned in Subsection 2.4 does not allow fully modular DFT models. In particular the dependent inputs of FDEP gates must be basic events and the spare-inputs of a spare gate are also required to be basic events (see [9, Sections 4.2.2 and 4.2.3]). In our approach we allow these inputs to be complex events (i.e. gates) as well as basic events, although spares are still required to be independent subtrees, as defined in Definition 4.2.

Before tackling the problem of making the DFT formalism more modular we must ask ourselves whether this is worthwhile. In particular we must answer the question whether is it useful to use independent sub-trees instead of basic events as spare inputs of a spare gate. Consider a computer system that simply consists of two processors, one primary processor and one spare. Because switching between processors must be very quick the spare processor will be running in a stand-by mode when it is not yet active. Figure 4.1 shows how we can model this computer system as a simple DFT. This DFT can be solved using the Galileo tool, because the spare-input of the spare gate is indeed a basic event. However, what if we want to increase the fault tolerance of our system by replacing the single processors by two processors running in parallel. We could model this new computer system using the DFT in Figure 4.2. There

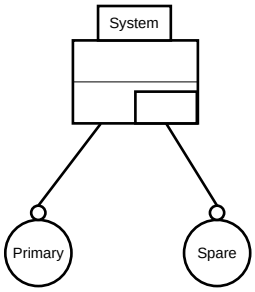

Figure 4.1: Example of a dynamic fault tree.

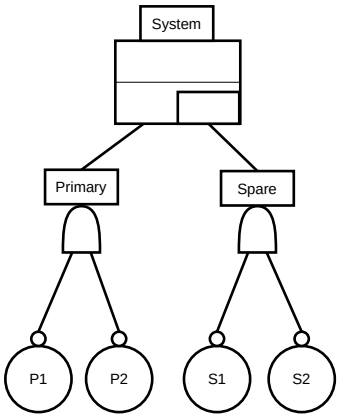

Figure 4.2: Example of a modular dynamic fault tree

is no reason why we wouldn't want to analyze such a DFT, so it is definitely worthwhile to expand the DFT formalism to allow such DFT.

In our approach the inputs of spare gates may be any DFT element. *We do, however, restrict the inputs to spare gates to being independent sub-trees.* For instance in Figure 4.2 the sub-tree consisting of *Primary*, *P*1 and *P*2 is independent since it is only connected to other elements via its top node *Primary*. The same goes for the sub-tree of element *Spare*.

Another way in which DFT are more modular in our approach is that *we allow complex events to be dependent via FDEP gates*. This means that the dependent inputs of an FDEP gate are not restricted to being basic events.

Allowing complex events to be used as spares does cause some problems in the interpretation of DFT models. These problems all revolve around the concept of dormancy. We have already mentioned that spares can be dormant or active and that they are activated by spare gates when necessary. For basic events dormancy is simple: the rate at which they fail simply changes. For complex events activation is more involved, but still manageable. We will discuss the meaning of dormancy for complex events used as spares in the next section.

4.3 Activation and Dormancy

The introduction of spare gates to the DFT formalism has also introduced the concepts of activation and dormancy. Recall that when the primary component of a spare gate fails it *activates* its first available spare and before it is activated this spare is *dormant*. A good example of this is the spare tire in the example DFT described in Section 2.3. In this case the concept of dormancy is simple, because the spare tire is modelled as a basic event: the spare tire simply fails at a lower rate when it is dormant. In the previous section we have seen, however, that in our approach we also allow independent subtrees to be spares and that means we must define the concept of dormancy for independent subtrees.

To decide how to model dormancy for independent subtrees we must consider what a DFT actually models and especially what the difference is between basic events and gates. In DFT models basic events model the behavior of physical objects. Gates on the other hand model the consequences of other events happening and never the behavior of physical components. If we, for instance, look back on the example of a DFT for a road trip (see Section 2.3) we can see that basic events all model the behavior of physical objects (tires, the engine and the mobile phone) and gates model the consequences of the failures of their inputs. It can be argued that gates can model some *complex* component (such as the car in our road trip example), but we will always find that every one of its physical subcomponents is modelled by some basic event in the subtree under this gate. The only gate that might be considered to model a physical device is the spare gate, since it must physically detect the failure of its primary component and then switch operation to one of its spares. If we, however, want to incorporate this physical aspect of a spare gate in our DFT we must once again use a basic event to model its possible failure.

We therefore define the dormancy of an independent subtree as the dormancy of its basic events. When an independent subtree, used as a spare, is activated all its basic events are activated. If we look at the DFT in Figure 4.3 we see that when component 1 fails component 2 is activated. The activation of component 2 simply means that both basic events (A2 and B2)

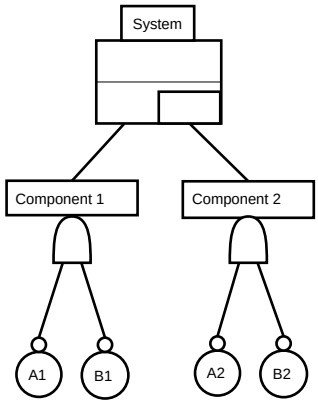

Figure 4.3: An example of a dynamic fault trees with components.

are activated. The behavior of the AND-gate doesn't change. Because A2 and B2 are warm basic events they may fire even when they are dormant. If they both do then component 2 fires before component 1, but this is proper behavior for warm basic events.

Things get interesting when we look at nested spare gates (i.e. spare gates that are part of a spare of another spare gate). For instance in Figure 4.4 the spare component (as well as the primary component) consists of a spare gate with a primary basic event and a spare. Should component 1 in Figure 4.4 fail then *P2* will become active but for *S2* nothing will

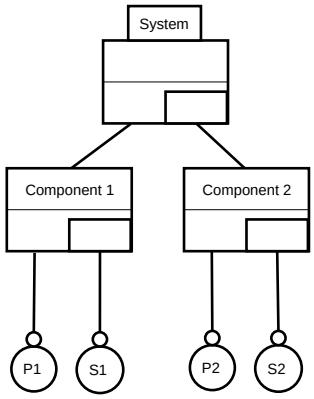

Figure 4.4: An example of a dynamic fault trees with nested spare gates.

change. It stays dormant until *P2* fails as well. Should *P2* fail before component 1 does (this is possible since it is a warm basic event) then *S2* also stays dormant. This means that spare gates do behave differently when dormant or active. The difference is that active spare gates activate their spares, but dormant spare gates don't. In this approach if an independent subtree is dormant then everything inside the independent subtree is also dormant and when an independent subtree becomes active everything inside the independent subtree, except for the spares, becomes active. The activation of spares is still regulated by the spare gates that share them.

We have now clearly defined dormancy for independent subtrees. Only basic events and spare gates have different active and dormant behaviors, other gates have only one behavior. For subtrees that are not independent dormancy is not well-defined because it becomes unclear when certain basic events and spare gates activate. Thus we restrict the inputs of spare gates to being independent subtrees.

4.4 Notation

In this section we discuss the notations used in the rest of this chapter.

In general DFT events (modelled as either gates or basic events) evolve through four stages. At first the event hasn't been activated yet. We call this the dormant stage. In the case of warm or hot events the event may move from its dormant stage to its firing state (see below). Activation of an event can be seen as the switching on or taking into use of a component. After activation the event is fully operational. The active stage of a DFT event is usually comprised of multiple IOIMC states. Eventually the conditions may be right for the event to fire (for instance, k of the m input-events of a k/m gate may have fired). In this third stage, the firing stage, the event will fire and so it will get to its last stage in which the event has fired. These four stages are shown in Figure 4.5.

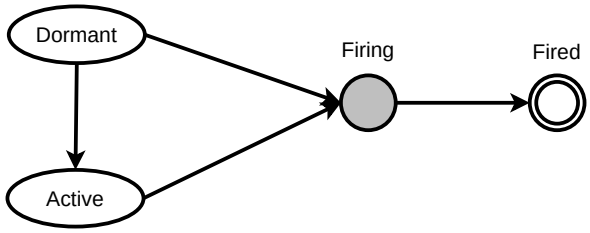

Figure 4.5: The life cycle of a DFT event.

We will see that the firing stage of the DFT elements life-cycle always corresponds with one IOIMC state. This state will be colored gray in the IOIMC models shown in this section and will be referred to as the *failed* state or the *firing* state. In the same way the fourth stage also corresponds with a single IOIMC state. This state will be denoted with a double circle in the IOIMC models of this section and will be referred to as the *fired* state. IOIMC states that correspond with the associated DFT element being operational (in the dormant or active stage) are simply denoted as white circles. They will be referred to as *operational* states. The dormant stage is not depicted in any special way in the IOIMC diagrams. Basic events are only dormant (first stage) in their starting state (already denoted with a dot in the middle). The dormancy of a gate depends totally on the dormancy of its inputs. Therefore we do not see a dormant stage in the IOIMC models of gates, except in that of the spare gate. The way activation has been modelled using IOIMC is described in detail in Subsection 4.6.2. Lastly we must note that for the PAND-gate there is a fifth stage of operation in which the gate can no longer fire, no matter what happens. This is called the disabled stage and always consists of a single state denoted with an **X**. Figure 4.6 shows the notations described in this section.

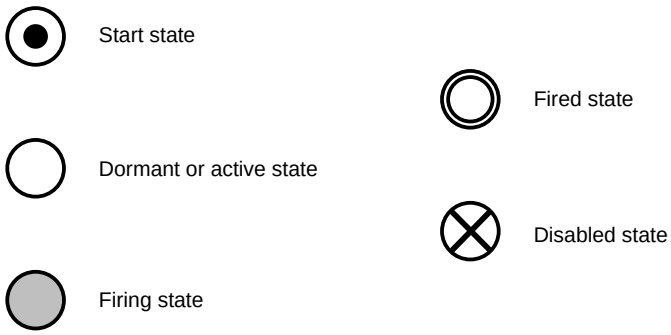

Figure 4.6: Different IOIMC states.

We will also use the following naming convention for interactions in the IOIMC models. $f(A)$ denotes the action of event A firing (or the failing of component A). $a(A, B)$ denotes the activation of the spare A by spare gate B. $a(A)$ denotes the activation of the independent subtree with top-node named A. All IOIMC models given in this thesis are fully input-enabled as described in subsection 3.2. **For the sake of simplicity we have sometimes left out those input-actions which have no effect on the state of the IOIMC (in other words, we have left out input-actions from a state to itself)**.

4.5 IOIMC models

In this section we will give some examples of IOIMC models of DFT elements, based on their names, their parents, their inputs and their delay-rates (in the case of basic events). This information can be directly derived from the formal definition (c.q. Definition 4.2) of the DFT containing the DFT element. The formal translation of the elements in a DFT to a community of IOIMC is given in Section 4.7. The IOIMC models described in this chapter can be found in full in Appendix B.

4.5.1 Basic Events

There are three different basic events. In their operational stage basic events all behave the same way: after an exponentially distributed delay they fail, signalling this failure to the rest of the IOIMC community.

Cold basic events cannot fail before activation. The IOIMC model of the cold basic event is shown in Figure 4.7. This particular basic event is named A and has a delay rate of λ.

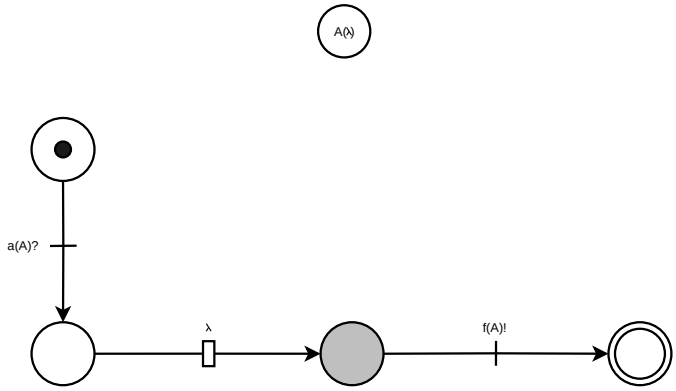

Figure 4.7: IOIMC model of a cold basic event: $(BE, 0, \lambda, 0)$

Warm basic events can fail before activation, but they do so at a different (usually reduced) rate. The IOIMC model of a warm basic event with name A, active delay rate λ and dormant delay rate μ is given in Figure 4.8.

Hot basic events can also fail before activation, and they do this at the same rate as after activation. The IOIMC model of a hot basic event with name A and delay rate λ is given in Figure 4.9.

4.5.2 OR gate

An OR gate fires when one of its inputs fires. In Figure 4.10 we see the IOIMC model of the OR gate with two inputs. Note that the IOIMC model has no transition labelled with an activation action. This means that the activation action is not an input-action for this IOIMC. This is also the case for AND, K/M and PAND gates. The generalization of the OR, AND, K/M, PAND and spare gates can be found in Appendix B.

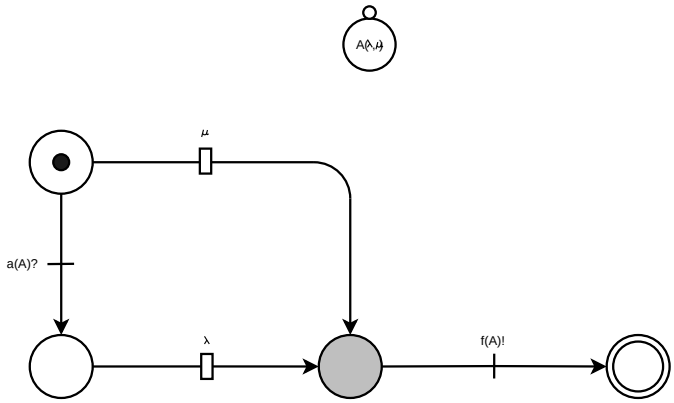

Figure 4.8: IOIMC model of a warm basic event: $(BE, 0, \lambda, \mu)$

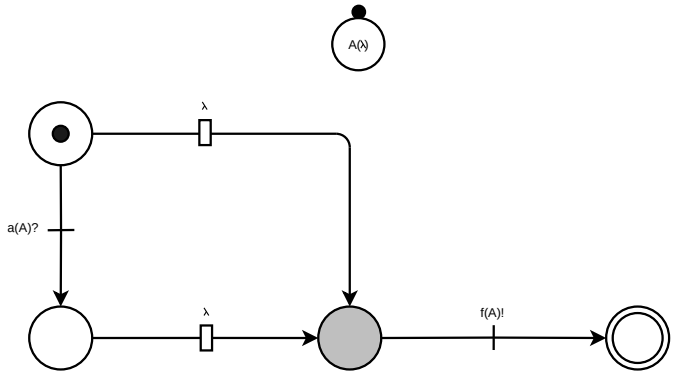

Figure 4.9: IOIMC model of a hot basic event: $(BE, 0, \lambda, \lambda)$

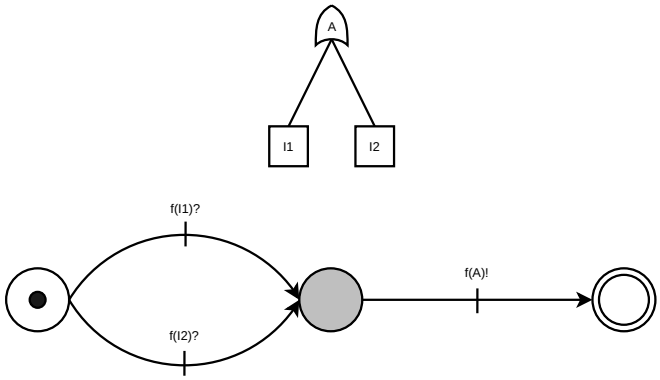

Figure 4.10: IOIMC model of the OR gate: $(OR, 2)$

4.5.3 AND gate

Figure 4.11 shows the IOIMC model of the AND gate with two inputs $I1$ and $I2$. The AND-gate fires when both its input events have fired.

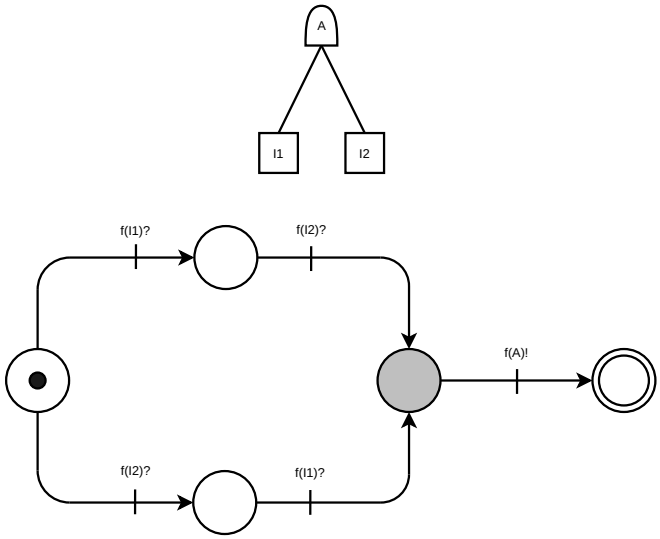

Figure 4.11: IOIMC model of the AND-gate: (*AND*, 2)

4.5.4 K/M-gate

A 2/3-gate with three inputs $I1$, $I2$ and $I3$ is depicted in Figure 4.12. It fires when at least two of its three inputs have fired.

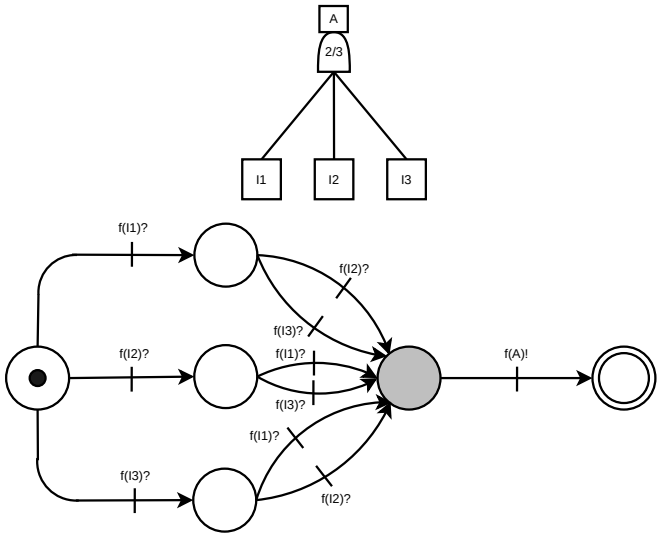

Figure 4.12: IOIMC model of the voting gate: (*VOTING*, 3, 2)

4.5.5 PAND gate

The PAND gate fires if all its inputs fire in left to right order. If the inputs fire in the wrong order, the PAND gate moves to its disabled state and can then never fire. Figure 4.13 shows the

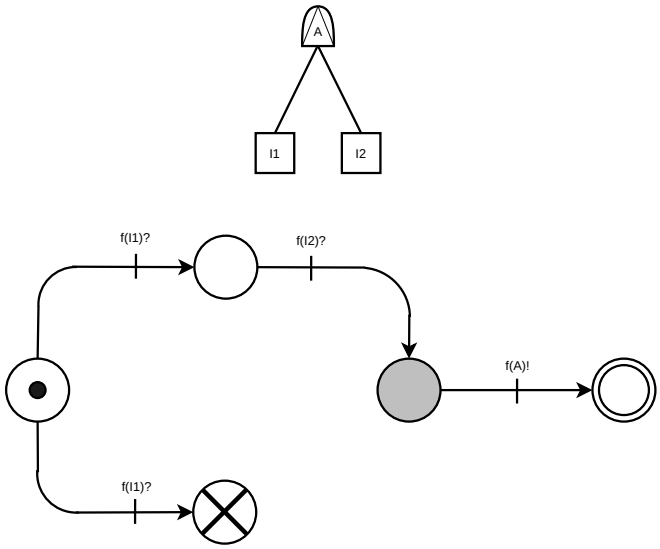

Figure 4.13: IOIMC model of the PAND gate: (*PAND*, 2)

IOIMC model of the PAND gate with two inputs. Note that in our approach inputs cannot fire at the same time. If the situation occurs that two DFT elements are ready to fire at the same moment, then the order in which they fire is chosen non-deterministically (see Section 4.6.5 for a discussion of this design choice).

4.5.6 Spare Gate

Figure 4.14 shows an IOIMC model of a spare gate named A with primary P and spare SP_P. This spare is shared between spare gate A and spare gate C. This model looks quite complicated but in fact it isn't. Note first of all that the spare gates shows dormant and active behavior in contrast to the other gates. Figure 4.15 shows a simplified view of the same model, where the dormant and active behavior of the spare gate have been split. In order to switch from dormant to active behavior the input action should be added from the dormant states to the appropriate active states. The dormant and active firing states can be merged, because when the spare gate is ready to fire it no longer matters whether it is dormant or active. The same goes for the fired state.

Looking at the simplified model we can see that, when dormant, a spare gate acts a lot like an AND-gate. This is no surprise considering a spare gate fails when its primary *a*nd all of its spares have failed. The difference with an AND-gate is that spares being activated by another spare gate has the same effect as that spare failing. The active behavior of a spare gate differs from the dormant behavior in that the spare gate will now try to activate spares when appropriate. The fact that dormant spare gates never activate their spares is in line with our interpretation of dormancy discussed in the Section 4.3.

In Appendix B the complete IOIMC model of a spare gate is given. Because a spare gate can have multiple spares and because all of these spares can be shared between any number of spare gates the behavior of a spare gate seems quite complex. However, the behavior of a

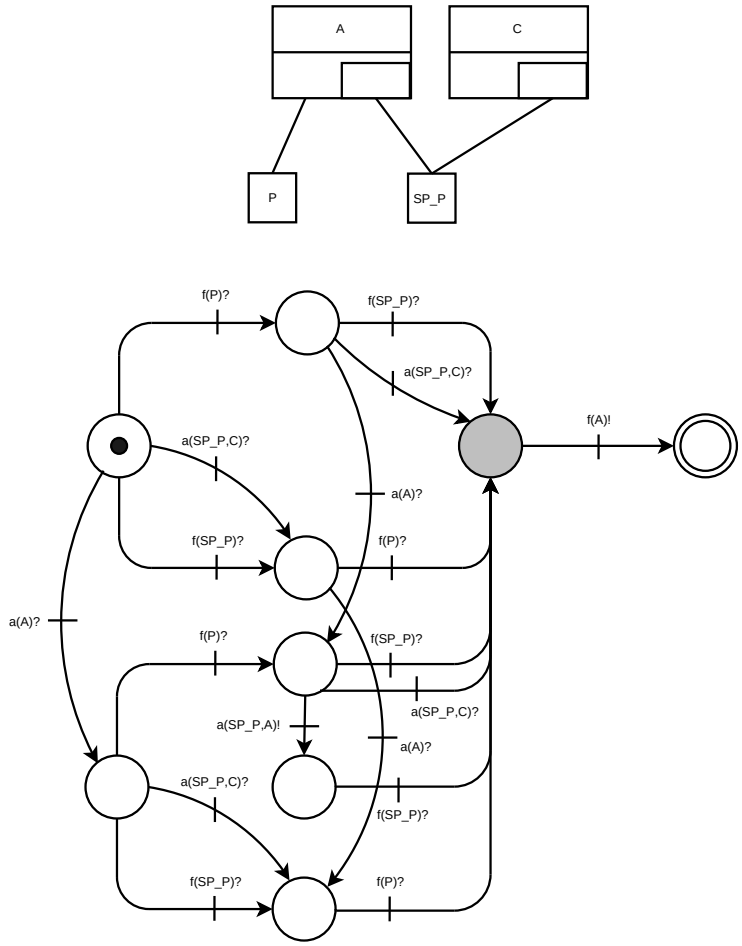

Figure 4.14: IOIMC model of the spare gate: $(SG, 2)$

spare gate can be reduced to four basic states. It can be either dormant or active and it can have a primary running or not. Figure 4.16 shows the general behavior of a spare gate. At first the spare gate is dormant and its primary component is operational. Three things may happen: the primary component may fail (although it is of course dormant), one of the (dormant) spares may become unavailable, either because it fails or because it is activated by another spare gate and finally the spare gate itself may be activated. A spare becoming unavailable doesn't really change the situation except of course that there is now one less spare available. When the spare gate is activated with its primary component still operational its behavior doesn't really change. Spares can still become unavailable and the primary can still fail. Notice that the spare gate does not activate the primary. Primary and spare gate however have the same activation signal so they will be activated at the same time. The difference between active and dormant behavior can be seen when the spare gate loses its primary component. When the spare gate is active it will activate the first of its available spares, but when the spare gate is dormant it won't. Notice also that at the moment the spare gate wants to activate its first spare any spare (including the first) may still become unavailable. This situation may give rise to the non-determinism discussed in Section 4.6.5. Finally a spare gate will fire in one of two situations: either the

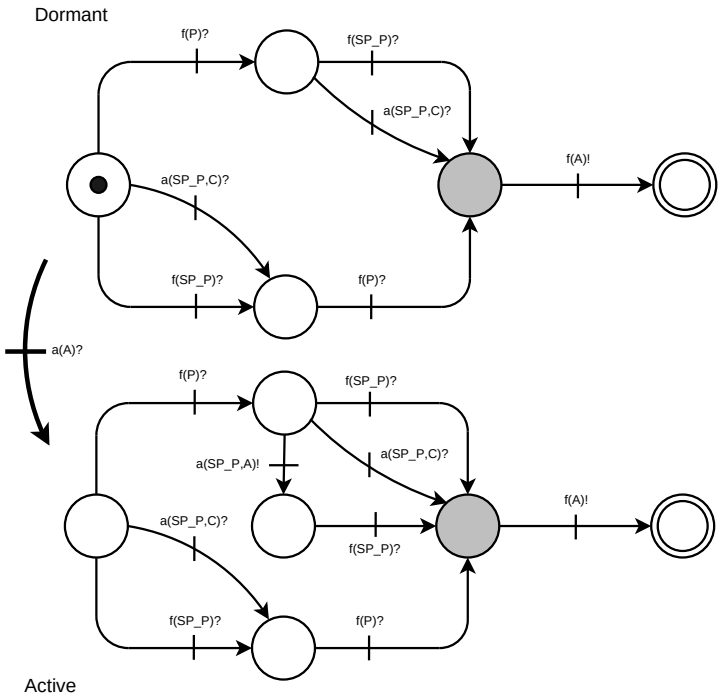

Figure 4.15: Simplified IOIMC model of the spare gate.

primary fails and no more spares are available or the spare gate doesn't have an operational primary and the last of its spares becomes unavailable (so the spare gate cannot activate this spare to become the new primary). Both situations can occur when the spare gate is dormant or active.

4.5.7 FDEP gate

The proper modelling of functional dependencies is achieved by using firing auxiliaries (FA). For some dependent DFT element a firing auxiliary governs when its firing action should be taken, namely when either the DFT element fires normally (i.e. as if it were not dependent) or when one of its trigger events fires. Figure 4.17 shows the firing auxiliary of a DFT element A which is dependent on two other DFT elements B and C. The action $f^*(a)$ corresponds to the normal firing of event A while action $f(a)$ models the actual firing of event A, whether normally or triggered by some other event.

So an FDEP gate in the DFT is modelled by one firing auxiliary IOIMC for each of its dependent inputs.

4.6 Design choices

The structure (i.e. syntax) of dynamic fault trees is defined formally in Section 4.1. In this section the behavior (i.e. semantics) of dynamic fault trees is defined by modelling them as input/output interactive Markov chains. To model DFT elements as IOIMC a number of

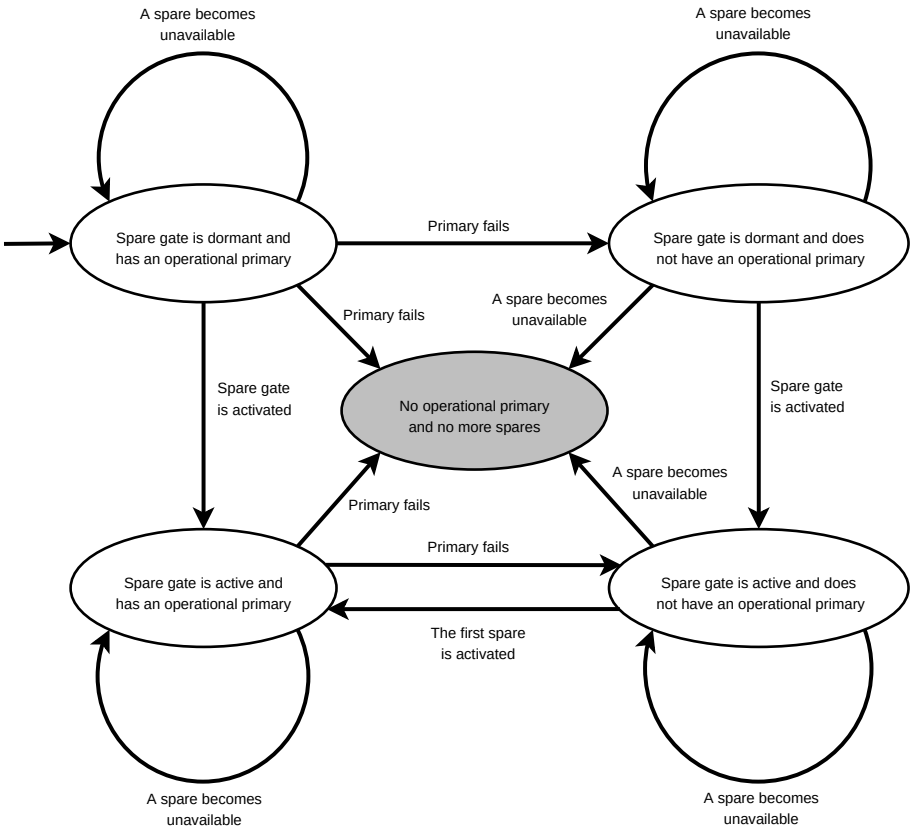

Figure 4.16: Schematic model of a spare gate.

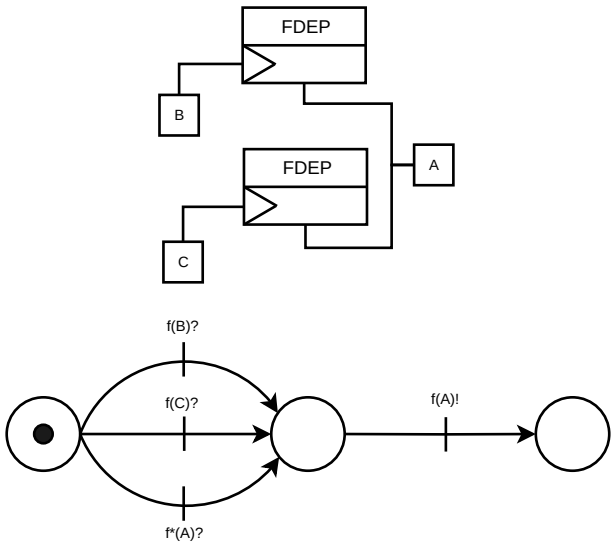

Figure 4.17: Firing auxiliary IOIMC model.

important design decisions need to be made. In particular we will look at the way the concepts of events firing and events being activated are modelled. In general every DFT is modelled as an independent subtree (which can possibly be used in another DFT), such that every DFT may be activated, may fire or may be dependent on other events.

4.6.1 Firing

The firing of events is the basis of the DFT formalism and is also the simplest to model using IOIMC. Every event, whether basic or complex may at some point fire. This is modelled by an interactive transition with an output action $f(A)!$ for an event named A. DFT elements that have event A as input respond to it firing by having input-transitions labelled $f(A)?$. So, the output action $f(A)!$ is used to signal the firing of event A and any interested DFT element may respond to this firing by having an input transition labelled $f(A)?$.

4.6.2 Activation

Activation is more complex than firing. Although only basic events and spare gates have an activation input-action (see Section 4.3), *all DFT elements are defined to have an activation signal*. Elements other than basic events and spare gates simply do not use this signal (it is not in the action signature of their IOIMC). For basic events and spare gates that are not used as spares activation is simple: they are activated when their parents are activated. This means that they have the same activation-signal as their parents. The top element of a DFT always has its own activation signal.

For spares activation is more complicated. A shared spare can be activated by several different spare gates. We have chosen the following approach to modelling spare gate-, primary-, and spare behavior:

- Primaries, which are unique for each spare gate, are activated together with their spare gate.
- Spares can be activated by any of their sharing spare gates, each having a *different* spare-activation signal.
- A spare gate can activate any of its spares. A spare gate also responds to the activation *by other spare gates* sharing one or more of its spares.

The way these elements communicate is shown in Figure 4.18.

The activation auxiliary (AA) model is used to allow a spare to be activated by several different spare gates. As inputs the activation auxiliary has the activation signals of the spare gates and as output it has the activation signal for the DFT element used as a spare. After it receives one of the activation signals from one of the spare gates the AA activates the spare. In Figure 4.19 we see the activation auxiliary for a spare which is shared by two spare gates.

The most interesting aspect of this communication is the fact that each spare gate has its own unique signal to activate the shared spare. A simpler approach is to have a single activation signal for spare component S, but apart from the difficulty of having two identical output-actions ($a(S)!$ for both the spare gates) the two spare gates *must* signal to each other that they

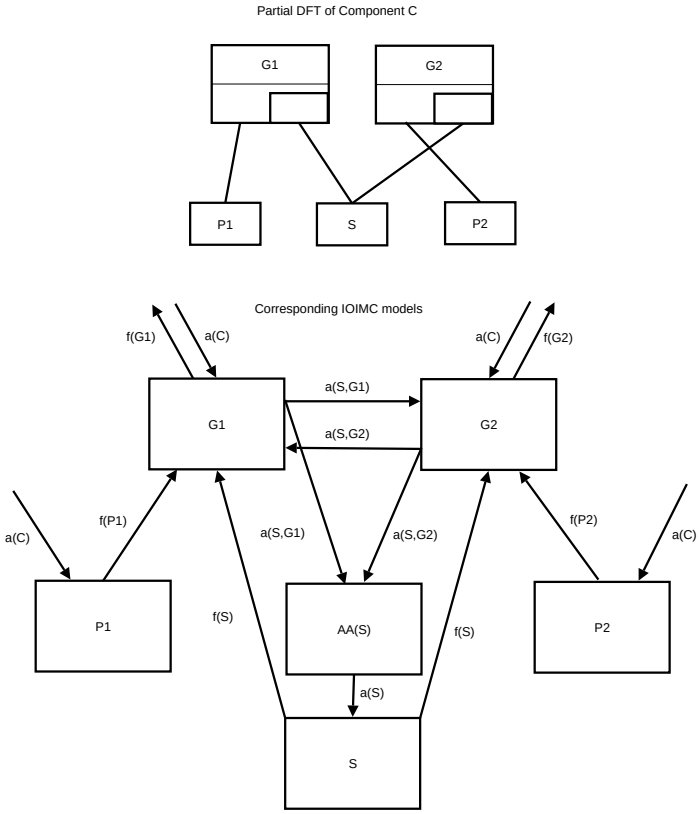

Figure 4.18: Communication between spare gates, primaries and spares

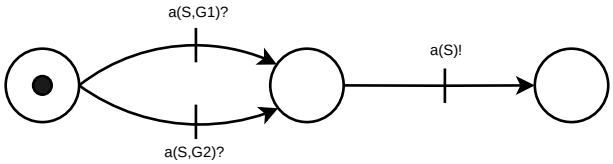

Figure 4.19: Activation auxiliary IOIMC model.

have activated the spare. This is of course only possible if these two signals are distinct. We will also see that the approach we have chosen deals very nicely with the situation when both primaries fail at the same time (due to a shared functional dependency). In this case one of the two spare gates will activate the spare, while the other fires. The spare gate that gets the spare is chosen non-deterministically as intended (see Subsection 4.6.5).

We have seen that DFT elements that are not used as spares get the same activation signal as their parents. This is of course only possible if each parent of such a DFT element has the same activation signal. The restriction that spares must be independent subtrees (see Section 4.2) ensures that this is the case. If two parents of a DFT element have different activation signals then obviously one of them must at least be part of a spare (or they are part of two different spares), which immediately means that this spare cannot be an independent tree and therefore

the DFT is not valid.

4.6.3 Time

In our IOIMC models of DFT elements time is modelled by using Markovian transitions. The only models containing Markovian transitions are the basic events, because these model the fact that there is a delay before a basic event happens. These delays are exponentially distributed to make sure that the models adhere to the Markov property. Although it is not possible to directly use other distributions, they can be approximated using, for instance, phase-type distributions [28].

4.6.4 Auxiliary IOIMC models

In Section 4.5 we have seen that every IOIMC model has at least a firing output-action and some also have an activation input-action. This generic approach also requires to use only very local information when constructing an IOIMC model of a DFT element. This, however, does cause a problem when considering shared spares and dependent DFT elements.

A spare that is shared by n spare gates will not have a single activation signal, but rather n different, dedicated, activation signals. One for each spare gate. To ensure that the spare activates when one of the spare gates tries to activate it an activation auxiliary (see Subsection 4.6.2) is added which acts simply as a relay between the spare and its sharing spare gates. The use of the activation auxiliary is shown in Figure 4.20.

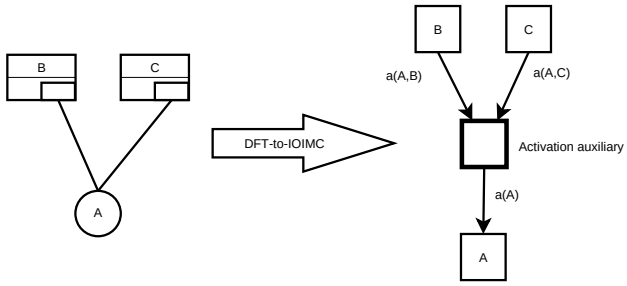

Figure 4.20: Example of how an activation auxiliary IOIMC is used to properly model spare-activation.

Dependent DFT elements fire in two different ways: either they fire normally by themselves or they are triggered by some DFT element. The firing auxiliary (see Subsection 4.5.7) models this behavior by outputting the DFT element's firing signal after it receives either the firing signal of one of the triggers or the firing signal of the dependent DFT element itself. This of course does mean that the firing signal of the dependent DFT element needs to be renamed. After renaming the signal and parallel composing the resulting IOIMC with the firing auxiliary the renamed firing signal can then be hidden. The use of the firing auxiliary is shown in Figure 4.21. In this figure the firing signal of basic event A is renamed to $f^*(A)$.

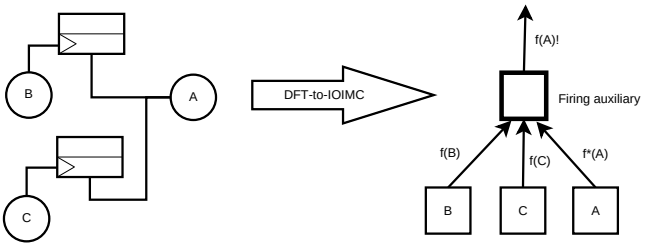

Figure 4.21: Example of how a firing auxiliary IOIMC is used to properly model dependency.

4.6.5 Simultaneity

When the trigger of an FDEP-gate fires all its dependent inputs also fire. This implies that both the trigger and the dependent events fire at exactly the same moment. However, the failure of these events is modelled by the sending of a number of different signals. For instance, in Figure 4.22, event A triggers events B and C. Their failure is modelled by the actions f_A, f_B and f_C.

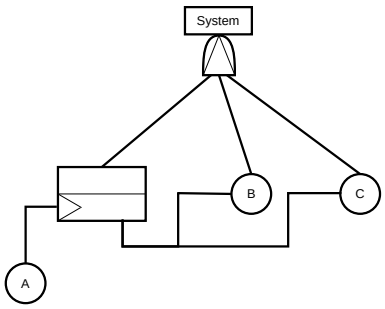

Figure 4.22: Example of a DFT with a functional dependency.

The fact that event A immediately triggers events B and C is modelled in the following way:

1. No time transpires between the failures of events A, B and C.
2. Event A occurs before events B and C.
3. Events B and C occur in a non-deterministic order.

To understand why we have chosen this way to model simultaneity caused by FDEP-gates we look at what an FDEP-gate models exactly. One of the uses of FDEP-gates is to model network elements in a computer system. For instance, in Figure 4.22, events B and C might model the failure of two remote computer systems connected to the main system via a but whose failure is modelled by event A. When the bus fails the remote computer systems immediately become unavailable to the main system which explains the first rule. The third rule may seem strange at first, because it seems logical to model events B and C to fail at exactly the same time. However, in real life systems, events rarely occur at exactly the same time and more importantly the occurrence of events is usually not *noticed* at exactly the same time. So

the third rule is based on the assumption that the main system will notice the unavailability of one of the remote computers before the other. Which one it notices first is not specified and the order is therefore modelled to be non-deterministic. Now the second rule seems debatable: why should the failure of the network be noticed before the unavailability of the two remote computers? The reasoning behind this ordering is based on the assumption that the occurrence of some event in a DFT is "noticed" simultaneously by all other events in the DFT. Whether this assumption is always correct is open to debate. It should be noted that the second rule also supports the notion of *causality*: the *cause* (in this case the trigger) always occurs before the *effect* (the dependent event).

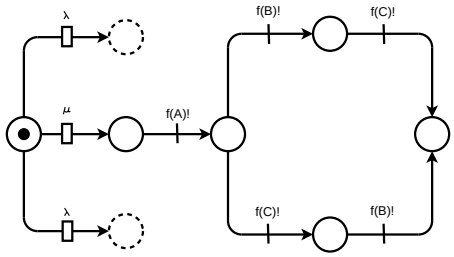

Figure 4.23: Partial IOIMC behavior of events A, B and C.

Figure 4.23 shows part of an IOIMC representing the events A, B and C observed in parallel. We can see that this IOIMC conforms to the three rules introduced above. When event A occurs (after an exponentially distributed delay with rate μ) first signal $f(A)$ is fired and then signals $f(B)$ and $f(C)$ in a non-deterministic order. This non-determinism has its consequences: in the running example the event *System* is modelled by a Priority-AND gate so the order in which events B and C occur determines whether the system fails. This is reflected in the IOIMC model of the behavior of this DFT shown in Figure 4.24 by the internal $f(A)$ transitions.

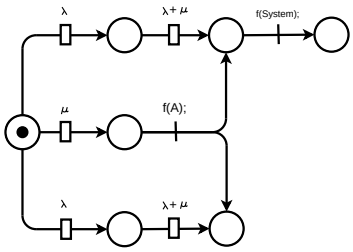

Figure 4.24: Simplified IOIMC behavior of the example DFT.

Because of the non-determinism the IOIMC model in Figure 4.24 is not isomorphic to a CTMC, but it can be interpreted as a Markov decision process (MDP) [33], a variation on CTMC. The MDP can then be analyzed to find upper and lower bounds for, for instance, the unreliability of the system.

4.7 Translation of a DFT to an IOIMC community

Before giving the translation of a DFT to an IOIMC community we give the formal translation of DFT elements to IOIMC models. The DFT elements are defined by their name, attributes (e.g. failure rates) and local surroundings in the DFT. In the DFT element descriptions all DFT elements have a name, a set of spare-parents P_{SP}, a set of regular parents P_R and a set of triggers T. The parents are the elements that have the element under discussion as an input. The spare-parents are spare gates that have the element under discussion as spare-input (so not as primary). All the other parents of an element are contained in the regular parents set. The triggers are DFT elements that this element is functionally dependent on. Furthermore basic events have one or two rates which define their probabilistic behavior and gates have a number of inputs contained in the vector I. K/M gates also have two parameters k and m which of course denote the threshold and total number of inputs for the gate.

In the IOIMC descriptions f_M denotes the firing signal of IOIMC M, a_M denotes its activation signal and A_M denotes the set of activation signals of the form $a_{M,x}$ for an IOIMC M. For the spare gate *pr* denotes the primary input while S denotes the vector of its spares. The function *Act* handles any operations needed to properly handle the activation of an IOIMC model based on the set of parents of the corresponding DFT element. The function *Dep* handles the triggering of a DFT element based on its set of triggers.

$$
\begin{aligned}
&[\![CBE(name, rate, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{CBE}(f_M, a_M, rate), T), P_{SP}, P_R) \\
&[\![WBE(name, rate_A, rate_D, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{WBE}(f_M, a_M, rate_A, rate_D), T), P_{SP}, P_R) \\
&[\![HBE(name, rate, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{HBE}(f_M, a_M, rate), T), P_{SP}, P_R)
\end{aligned}
$$

So, basic events all have a firing signal (f_M) an activation signal (a_M) and one or two rates. Just like all other IOIMC models activation must be dealt with based on the element's parents and dependency must be dealt with based on the element's triggers.

$$
\begin{aligned}
&[\![OR(name, I, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{OR}(f_M, a_M, \langle f_i \mid i \in I\rangle), T), P_{SP}, P_R) \\
&[\![AND(name, I, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{AND}(f_M, a_M, \langle f_i \mid i \in I\rangle), T), P_{SP}, P_R) \\
&[\![KM(name, k, m, I, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{KM}(f_M, a_M, k, m, \langle f_i \mid i \in I\rangle), T), P_{SP}, P_R) \\
&[\![PAND(name, I, P_{SP}, P_R, T)]\!] = \\
&\quad Act(Dep(\mathsf{IOIMC}_{PAND}(f_M, a_M, \langle f_i \mid i \in I\rangle), T), P_{SP}, P_R)
\end{aligned}
$$

The logical gates look much the same as basic events except that they also have a list of input signals. Note that lists are ordered which is important for the Priority-AND gate. As mentioned before the static gates all have an activation signal although none of them use this signal in their

IOIMC models. Why this is the case will become clear when the *Act* function is described below.

$$[\![SG(name, pr, S, P_{SP}, P_R, T)]\!] = \\ Act(Dep(\mathsf{IOIMC}_{SG}(f_M, a_M, f_{pr}, \langle (a_{s,M}, f_s : A_s \setminus \langle a_{s,M} \rangle) \mid s \in S \rangle), T), P_{SP}, P_R)$$

In the above equation ':' is used to prefix a list of actions with a single action. The spare gate is the most complex of the IOIMC models. Besides a firing and activation signal it also has the firing signal of its primary component. Furthermore the spare gate has a list of tuples. Each tuple corresponds to one of the spare gates spares and contains the spare's activation signal ($a_{s,M}$) and a list of disabling signals for the spare. The disabling signals consist of the failure signal of the spare (f_s) and the activation signals of the other spare gates sharing that spare (A_s). The spare gate may use the activation signal for a spare ($a_{s,M}$) to activate the spare.

$$Act(M, P_{SP}, P_R) = \\ \begin{cases} M & \text{, if } P_{SP} = \langle\rangle \wedge P_R = \langle\rangle \\ M[a_M \setminus a_p] & \text{, if } P_{SP} = \langle\rangle \wedge P_R \neq \langle\rangle \\ & \text{, where } p \in P \\ \textbf{hide } a_M \textbf{ in } M \| \mathsf{IOIMC}_{AA}(a_M, \langle a_{M,p} \mid p \in P_{SP} \rangle) & \text{, if } P_{SP} \neq \langle\rangle \end{cases}$$

The handling of activation is based solely on the parents of an element which are split into two sets: one containing spare gates that use the element as a spare and one containing other parents. There are three possibilities for activation of a DFT element. First, the element may be the top element of the DFT having no parents. In this case nothing needs to be done to the IOIMC model. Secondly, the DFT element may be a primary element of the DFT, that is to say, not a spare. In this case the IOIMC's activation signal is renamed to the activation signal of one of its parents (this is denoted as $[a_M \setminus a_p]$). This is also the reason all IOIMC models need to have an activation signal defined even if they do not use them. It is easy to see that these first two clauses will result in all the primary elements of a DFT having the same activation signal as the top node. This is in correspondence to the way we have chosen to represent activation (see Subsection 4.6.2). The choice of parent for the renaming of the activation signal is not important as all parents must have the top node as common ancestor. This leaves the activation of spares which is handled by the third clause. When an element is a spare of one or more spare gates, then it gets a number of dedicated activation signals from these spare gates. To cope with these multiple signals the IOIMC model of the DFT element is composed with an activation auxiliary (as discussed in Subsection 4.6.4).

$$Dep(M, T) = \\ \textbf{hide } f'_M \textbf{ in } M[f_M \setminus f'_M] \| \mathsf{IOIMC}_{FA}(f_M, f'_M : \langle f_t \mid t \in T \rangle)$$

Dependent DFT elements are composed with a firing auxiliary to make sure that the element also fires when one of its triggers fires. First, however, the firing signal of the DFT element must be renamed. This renamed signal is used as input to the firing auxiliary and after composition it can be hidden.

Above we have seen how to translate individual DFT elements. We can now translate an entire DFT to a community of IOIMC by applying this individual translation on each of its elements (except the FDEP-gates):

Definition 4.3 *Let $D = (V, in, l, T)$ be a DFT. The* IOIMCC-translation *of D is an IOIMC community C where:*

$$C = \{[\![\mathsf{details}_V(v)]\!] \mid v \in V \ \wedge \ type(l(v)) \neq FDEP\}$$

The function $\mathsf{details}_V$ *is defined below as well as the auxiliary functions P_{SP}, P_R and Trig for DFT D which give the spare parents, regular parents and triggers for a node of D respectively:*

$$\mathsf{details}_V(v) = \begin{cases} CBE(v, \lambda, P_{SP}(v), P_R(v), Trig(v)) & , \text{if } l(v) = (BE, 0, \lambda, \mu) \ \wedge \ \mu = 0 \\ WBE(v, \lambda, \mu, P_{SP}(v), P_R(v), Trig(v)) & , \text{if } l(v) = (BE, 0, \lambda, \mu) \ \wedge \ \mu \neq \lambda \\ HBE(v, \lambda, P_{SP}(v), P_R(v), Trig(v)) & , \text{if } l(v) = (BE, 0, \lambda, \mu) \ \wedge \ \mu = \lambda \\ OR(v, in'(v), P_{SP}(v), P_R(v), Trig(V)) & , \text{if } type(l(v)) = OR \\ AND(v, in'(v), P_{SP}(v), P_R(v), Trig(V)) & , \text{if } type(l(v)) = AND \\ KM(v, k, m, in'(v), P_{SP}(v), P_R(v), Trig(V)) & , \text{if } l(v) = (VOTING, m, k) \\ PAND(v, in'(v), P_{SP}(v), P_R(v), Trig(V)) & , \text{if } type(l(v)) = PAND \\ SG(v, \mathbf{head}(in'(v)), \mathbf{tail}(in'(v)), & \\ \quad P_{SP}(v), P_R(v), Trig(V)) & , \text{if } type(l(v)) = SPARE \end{cases}$$

$$P_{SP}(v) = \{w \mid v \in in'(w) \ \wedge \ v \neq \mathbf{head}(in'(w)) \ \wedge \ type(l(w)) = SPARE\}$$

$$P_R(v) = \{w \mid v \in in'(w) \ \wedge \ w \notin P_{SP}(v)\}$$

$$T(v) = \{w \mid \exists x \in V \cdot type(l(x)) = FDEP \ \wedge \ w = \mathbf{head}(in'(x)) \ \wedge \ v \in \mathbf{tail}(in'(x))$$

In the above $\mathbf{head}(l)$ *gives the first item of a list l and* $\mathbf{tail}(l)$ *gives all the items of a list l except the first item.*

Chapter 5

Compositional Analysis of DFT

In this chapter we will describe an analysis technique for dynamic fault trees based on the compositional semantics of DFT elements described in Chapter 4. This analysis technique is called compositional analysis. Compositional analysis is based on the idea of composing the IOIMC models of the elements of a DFT into a single IOIMC which can then be analyzed to find fault tolerance measures. Below we describe the four steps needed to find a single IOIMC representing the behavior of a DFT. This IOIMC can then be used to find a CTMC representing the behavior of this DFT (step 5) and then we can solve this CTMC to find fault tolerance measures such as the unreliability of the system described by the DFT.

5.1 Step 1: Translation

First we must convert the building blocks of the given DFT into IOIMC. Figure 5.1 shows the conversion of an example DFT to a community of IOIMC. On the right side of the picture we see the IOIMC corresponding to the DFT elements on the left. The actions used to signal the firing of events are drawn as arrows. To simplify the figures, the concepts of activation and triggering have been ignored in this running example.

In Figure 5.2 we see the IOIMC models of the PAND-gate D and the basic event A. The translation of DFT elements into IOIMC models has already been described in detail in Section 4.7

5.2 Step 2: Abstracting composition

In order to calculate fault tolerance measures for our running example we need to transform the community of IOIMC shown in Figure 5.1 into a single IOIMC. We do this by using parallel composition and abstraction. Figure 5.3 shows the parallel composition of the IOIMC-model of PAND-gate D and the IOIMC-model of the basic event A in which the action $f(A)$ is hidden. In other words Figure 5.3 shows the abstracting composition of the subset $\{D, A\}$ in the IOIMC community (see Section 3.2). The resulting IOIMC community is shown in Figure 5.4.

The order in which the IOIMC are composed has a big impact on the efficiency of compositional analysis. The composition of IOIMC D and A, for instance, has only eight states which can be reduced to 7 through aggregation (see step 3). The composition of IOIMC E and A on

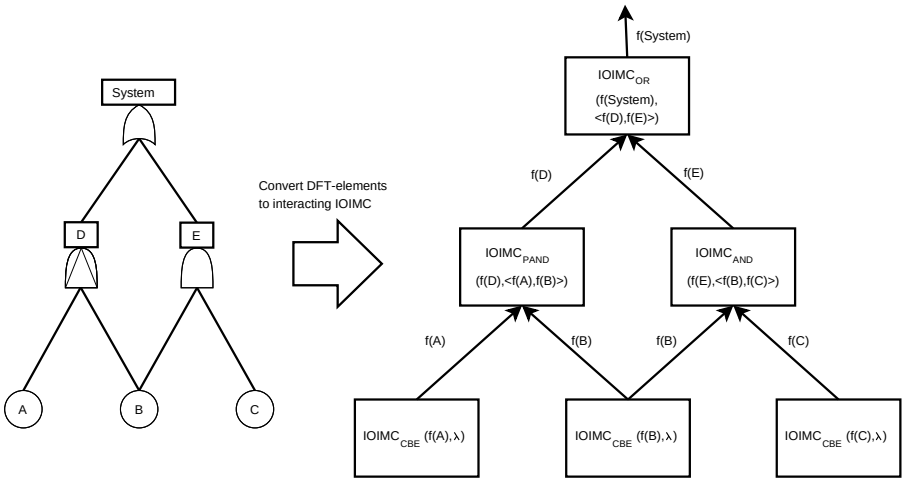

Figure 5.1: Conversion of a DFT into a community of IOIMC

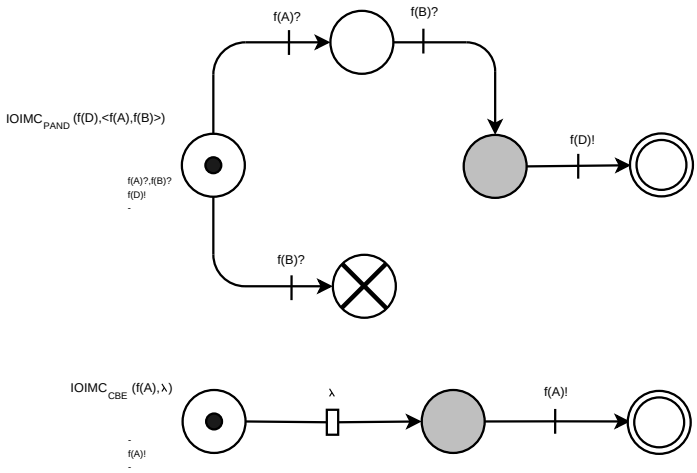

Figure 5.2: IOIMC models of two DFT elements. Their descriptions (as in Section 4.7) are given as well as their action signatures.

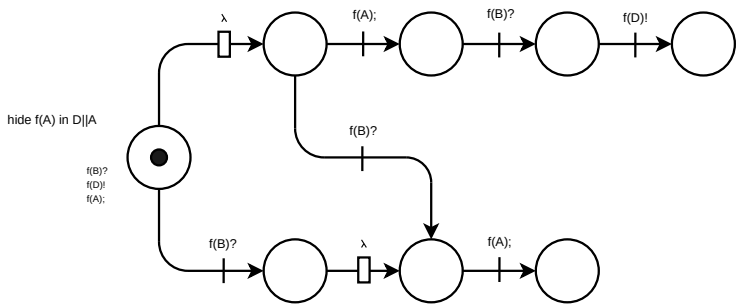

Figure 5.3: Parallel composition of a PAND-IOIMC and a basic event-IOIMC

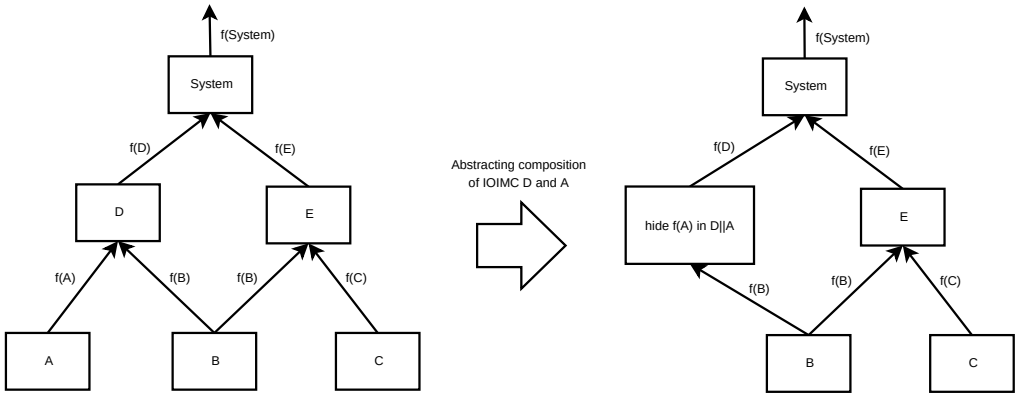

Figure 5.4: Result of the abstracting composition of IOIMC *D* and IOIMC *A*

the other hand has 15 states. This is caused by the fact that there is no synchronization in this composition (i.e. IOIMC *E* and *A* have no matching input/output actions). Ordering strategies for composition are further discussed in Subsection 8.3.2.

5.3 Step 3: Aggregation

The purpose of aggregation is to minimize the IOIMC models created by abstracting composition in step 2. Aggregating an IOIMC simply means creating a smaller IOIMC that is, in some way, equivalent with the original IOIMC. The choice of the equivalence used is crucial in this step. In this thesis we use the weak bisimulation equivalence to aggregate IOIMC models (see Section 3.4 for the definition of this equivalence and a discussion on why we choose it).

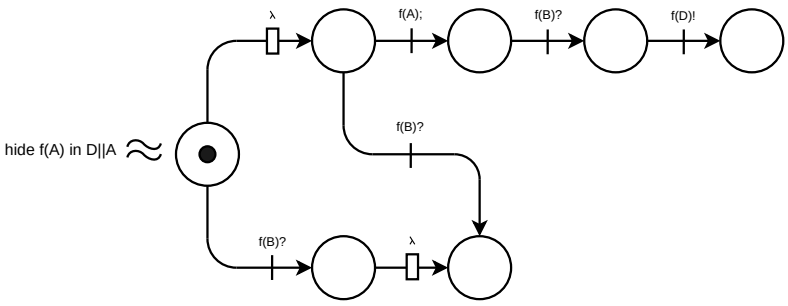

Figure 5.5: Result of aggregation using weak bisimulation

In Figure 5.5 we see the result of aggregating the IOIMC from Figure 5.3 using weak bisimulation. The only difference between the original IOIMC and the resulting IOIMC is the removal of one state in the lower right corner.

5.4 Step 4: Repetition

As mentioned before we need to have one single IOIMC to be able to calculate fault tolerance measures. In the examples we have seen so far we have transformed a DFT into six IOIMC models and then composed two of these to get five IOIMC models. By repeating steps two and three (composition and aggregation) we eventually get one single IOIMC model representing the behavior of the DFT. Figure 5.6 shows this IOIMC model.

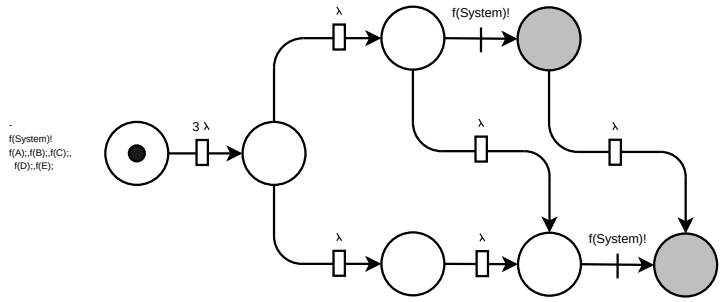

Figure 5.6: IOIMC representation of the running example. The grey states represent states in which the system has failed.

This IOIMC representation of a DFT can be reused in the compositional analysis of other DFT that have the example DFT as an independent subtree. An example of this analysis technique is shown in Section 6.1. If we, however, want to calculate fault tolerance measures such as unreliability for this DFT we need to transform the IOIMC into a CTMC.

5.5 Step 5: CTMC generation

The final step of compositional analysis is to find a CTMC representing the behavior of the DFT. An IOIMC can be interpreted as a CTMC if it has no interactive transitions, but the IOIMC we found for our running example still has two interactive transitions labelled $f(System)!$. But because we are now only interested in this IOIMC itself (and not in how it interacts with its environment), we can hide this signal and make the transitions internal. By then aggregating the IOIMC using weak bisimulation we abstract from these internal transitions and find an IOIMC which can be interpreted as a CTMC representing the behavior of the example DFT. This CTMC is shown in Figure 5.7.

The grey states in this CTMC are those states in the corresponding CTMC which are weakly bisimilar to the failed states in the IOIMC in Figure 5.6 after hiding signal $f(System)$. Note that when activation is taken into account the activation signal for the system also has to be hidden in order to find an IOIMC which can be interpreted as a CTMC.

We can now use CTMC analysis techniques to find fault tolerance measures for our example system. For instance, the unreliability of the example system for a mission-time of 1 equals 0.473 or 47.3%.

For certain DFT non-determinism will arise because of simultaneity. Because of this non-determinism it will not be possible to find a CTMC representation of these DFT, but this prob-

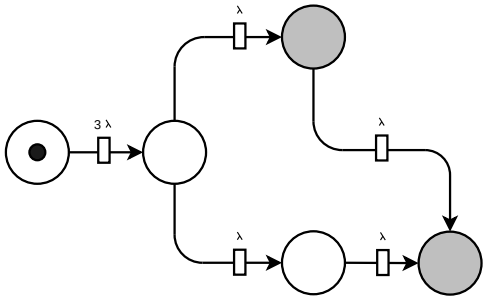

Figure 5.7: CTMC representation of the running example. The grey states represent states in which the system has failed.

lem can be avoided by altering the DFT or by converting the IOIMC to a different formalism such as MDP. For more information on this phenomenon see Subsection 4.6.5.

Chapter 6

Case Studies

In this chapter we will discuss three case studies. These case studies show that the compositional approach to analyzing DFT can be much more efficient than the traditional DIFTree approach. The case studies also give a good indication of when compositional analysis is expected to outperform DIFTree analysis.

We have analyzed the dynamic fault trees by first translating them to a number of Tipp processes using the DFT2Tipp tool and then using the Tipp tool [16] to perform parallel composition, abstraction and, partly, aggregation. This method of analysis is described in detail in Section 7. We have then used the Tipp tool to find the unreliability of the DFT for a certain mission time. We have compared the performance of this approach with the performance of the Galileo tool, which implements the DIFTree approach (see Section 2.4) by comparing the size of the generated IOIMC and CTMC respectively.

6.1 The Hypothetical Cascaded P-AND System case study

The first case we look at is a simple case study taken from [6] which is designed to show the shortcomings of the DIFTree approach to analyzing dynamic fault trees. Figure 6.1 shows the Hypothetical Cascaded P-AND System (HCPS). All basic events are cold and have a failure rate of 1 failure per time unit. Because the DFT has a dynamic top node the order in which all the basic events fail must be taken into account when transforming the DFT into a CTMC using the DIFTree approach. The resulting CTMC is therefore quite large (4113 states and 24608 transitions).

6.1.1 Compositional analysis

The HCPS DFT consists of two P-AND gates and three AND gates each with four identical basic events. In fact the three AND gate modules (consisting each of an AND gate and four basic events) are completely identical. We will use this fact to improve the efficiency of our analysis. To do this, one such module will be translated into an IOIMC and this IOIMC will be reused three times. Figure 6.2 shows the AND-gate module.

Note that the top node is called T instead of A, C or D. After transforming the module into an IOIMC the correct models can simply be found by renaming the IOIMC's firing and

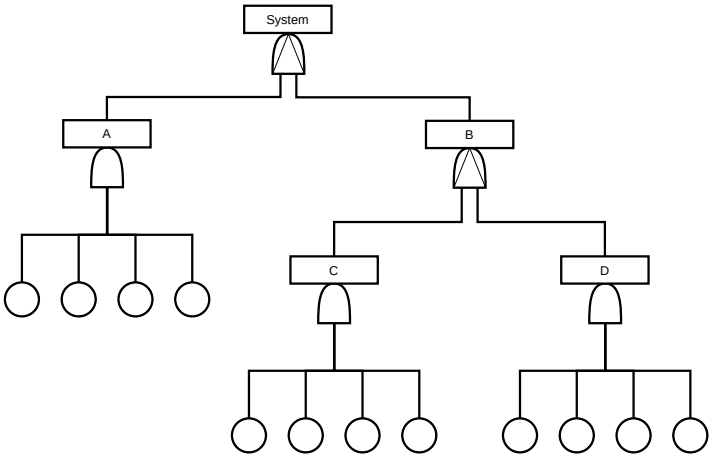

Figure 6.1: DFT of the hypothetical cascaded PAND system.

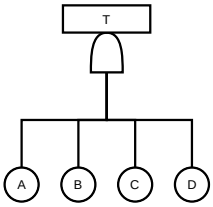

Figure 6.2: DFT of the component.

activation signal. This is an important advantage of the IOIMC formalism, where the IOIMC model of a DFT can be reused in the compositional analysis of other DFT. It is not possible to reuse the CTMC generated by the DIFTree approach.

First the DFT elements in the module are translated into IOIMC models which together form an IOIMC community (Step 1 of compositional analysis, see Chapter 5). This community is shown in Figure 6.3. After this translation compositional aggregation is used to find a single IOIMC which represents the behavior of the module.

Figure 6.4 shows the result of parallel composing IOIMC A and T, hiding action $f(A)$ and aggregating the result using weak Markovian bisimulation. This corresponds to steps 2 and 3 described in Chapter 5.

We now continue the compositional aggregation until a single IOIMC is found (Step 4 in Chapter 5). Figure 6.5 shows this IOIMC. When studying this IOIMC in detail we can see that it correctly models the behavior of the AND-gate module. After activation of the cold basic events we find that the first failure of a basic event occurs at four times the rate of a single failure. This models the fact that any of the four basic events may fail first and, in this module, it is unimportant which one fails first. The second failure then occurs at a three times the rate of a single event and so on until finally the AND gate itself fails.

We can now translate the entire HCPS DFT by reusing the IOIMC given that some renaming is done. Then compositional analysis (see Chapter 5) is used to find a CTMC representing the behavior of the hypothetical cascaded P-AND system. This CTMC is then analyzed to find

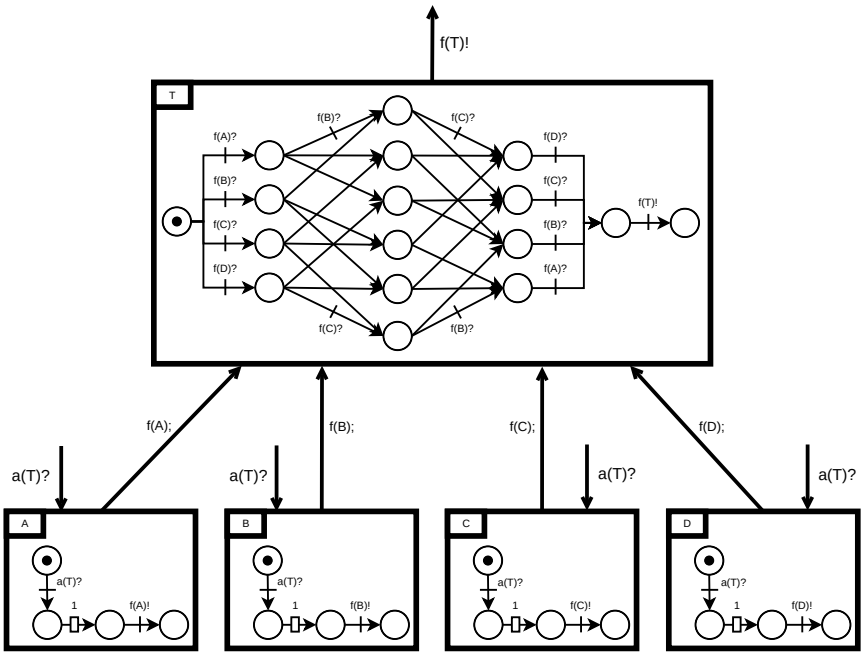

Figure 6.3: IOIMCC of the component.

the unreliability of the whole system.

6.1.2 Results

In table 6.1 we find the results of the HCPS case study. Note that there is no timing information as the compositional analysis was partially performed manually (see Section 7).

Approach	Maximum number of states	Maximum number of transitions	Unreliability (Mission-time = 1)
DIFTree	4113	24608	0.00135
Compositional	156	490	0.00135

Table 6.1: Results for the HCPS case study.

The DIFTree result is the size of the state space of the continuous-time Markov chain generated directly from the DFT. The result for the compositional approach is the largest appearing IOIMC during compositional aggregation. We also see that both analysis techniques give us the same unreliability for a mission-time of 1 time unit. In Figure 6.6 we see the sizes of the IOIMC for different steps in the compositional aggregation process.

We can see that the CTMC generated by the Galileo tool using the DIFTree approach is more than 25 times as large as the largest IOIMC appearing during compositional analysis. This large difference will have a serious effect on the performance of the tool in calculating measures such as unreliability for the DFT. This large difference is caused by the fact that

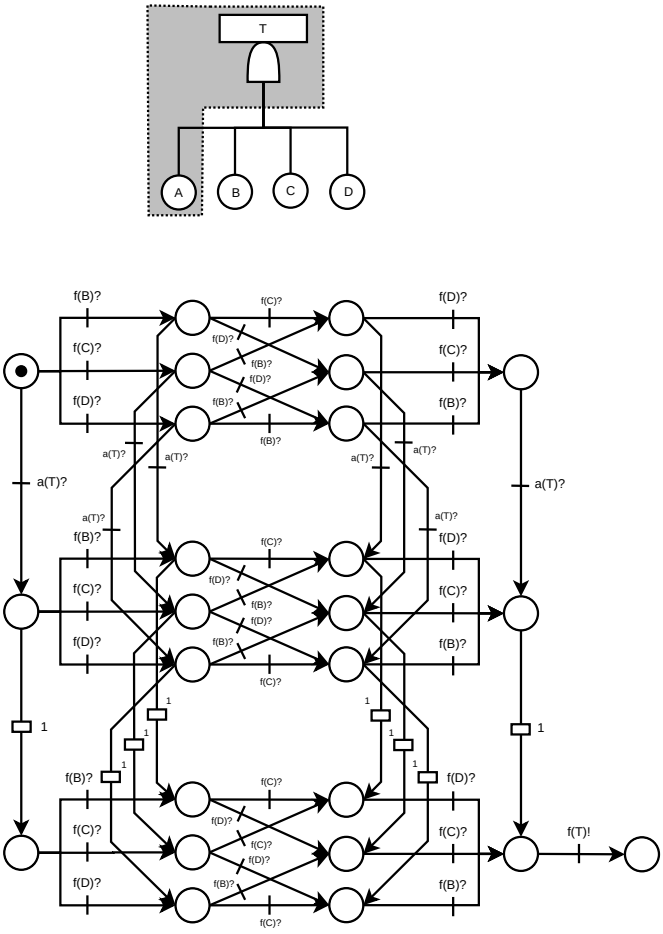

Figure 6.4: IOIMC of A||T.

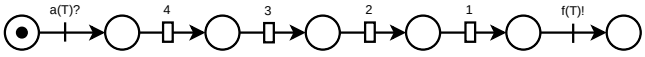

Figure 6.5: IOIMC representation of the module.

Galileo cannot use any modularization in its analysis. The reason for this is that the top node ‘System’ of the DFT is a PAND gate. Furthermore the DFT only fails when all of its twelve basic events fail (in the correct order). So to find the CTMC for the HCPS DFT Galileo must consider all the different orderings of the twelve basic events. In compositional analysis we make use of two characteristics of this DFT to minimize the appearing IOIMC models: The four basic events under each AND gate may fail in any order and all basic events have the same failure rate. This effect can be seen very clearly in the IOIMC representation of the AND-module shown in Figure 6.5 which has only 7 states.

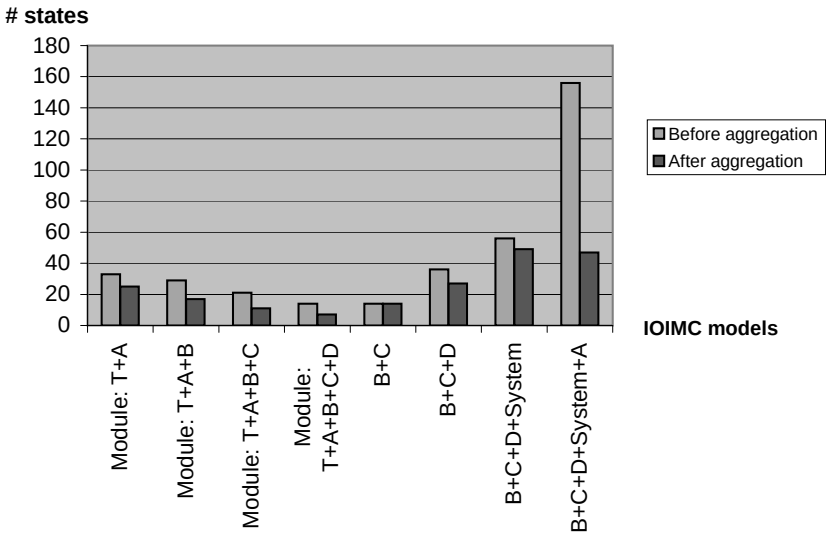

Figure 6.6: State space chart.

6.2 Cardiac Assist System

The second case study is based on the Cardiac Assist System (CAS) introduced in [30]. Specifically it considers a hypothetical variation on the CAS introduced in [6]. The DFT of the Cardiac Assist System (CAS) is shown in Figure 6.7.

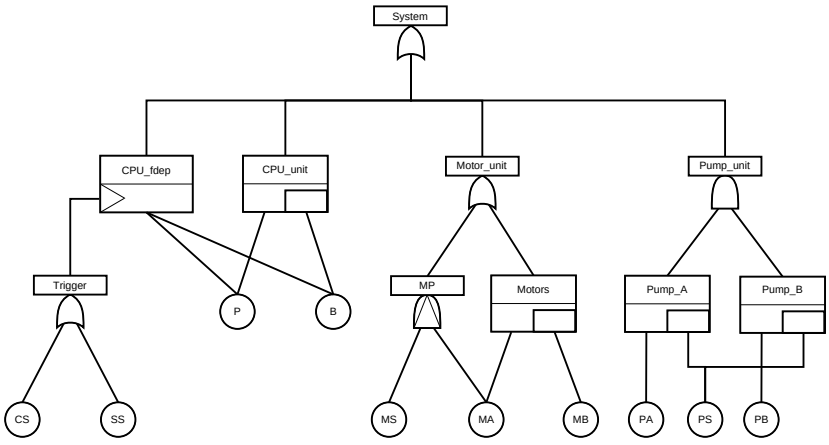

Figure 6.7: DFT of CAS.

The CAS consists of three separate modules: the CPU unit, the motor unit and the pump unit. These correspond to the CPU, motor and pump in the physical cardiac assist system that is modelled by the DFT. There are two CPU's: a primary (P) and a warm spare (B). Both are functionally dependent on a cross switch (CS) and a system supervision (SS), which means that the failure of either these components will trigger the failure of both CPU's. There are also two motors: a primary and a cold spare. The switching component that turns on the spare motor

when the primary fails is also subject to failure, but this failure is only relevant if it occurs before the failure of the primary motor. Finally, there are three pumps: two primary pumps running in parallel and a cold shared spare pump. All three pumps must fail for the pump unit to fail. Table 6.2 shows the failure rates of the different components:

Component	Failure rate
CS	0.2
SS	0.2
P	0.5
B	0.5
MA	1
MB	1
MS	0.01
PA	1
PB	1
PS	1

Table 6.2: Failure rates for the CAS case study

6.2.1 Compositional analysis

Using compositional analysis we found the aggregated IOIMC representations of the CPU, Motor and Pump unit which had 27, 11 and 6 states respectively. If we now wish to solve the DFT completely we must use compositional aggregation to combine the four remaining IOIMC (three IOIMC representations of the units and one IOIMC representation of the top OR-node).

The largest IOIMC that appears is the parallel composition of the IOIMC representing the combined behavior of the top node 'System', the CPU unit and the Motor unit and the IOIMC representing the behavior of the Pump unit. This IOIMC has 808 states and 3660 transitions. If we compare this with the results of modular analysis using the DIFTree approach we see that the Galileo tool generates three small Markov chains (between 4 and 6 states) and then uses binary decision diagram techniques (since the top node 'System' is a static gate) to very efficiently find the unreliability of the DFT.

To find out why compositional analysis performs so poorly for this case study, we take a closer look at the IOIMC representation of the CPU unit which has 27 states. The corresponding Markov chain generated by Galileo only has 4 states. Figure 6.8 shows the beginning of the IOIMC for the CPU unit after compositional aggregation.

The reason that we get such a large IOIMC is that the weak bisimulation used by Tipp tool to aggregate the IOIMC is not weak enough. For instance, Tipp tool does not differentiate between input and output actions and therefore it does not apply the maximal progress assumption (see Section 3.3) to output actions. If we look at Figure 6.8 we see that the Markovian transition from state *A* to state *D* could simply be ignored under the maximal progress assumption. Now let's consider state *B* in Figure 6.8. Because there is a firing transition from state *A* to

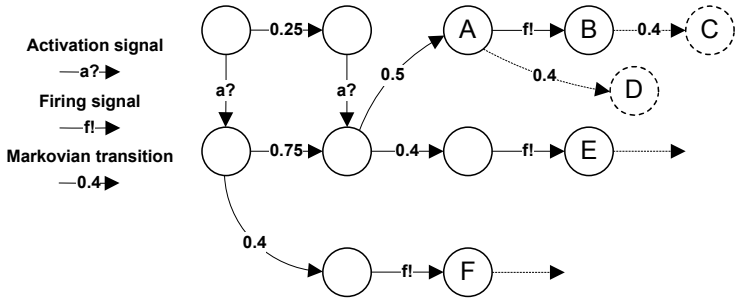

Figure 6.8: Partial IOIMC for the CPU unit.

state *B* we know that the CPU has failed in state *B*. We also see that there is a Markovian transition from state *B* to state *C*, but we are not interested in this behavior because the CPU has already failed. In fact we are not really interested in any behavior that can never lead to an output transition and we could aggregate such states into a single 'uninteresting' state. This notion of 'interesting' and 'uninteresting' behavior could lead to a new, weaker, bisimulation for IOIMC. More information on this notion can be found in Subsection 8.3.1.

To investigate the effect of this weaker bisimulation we have used it to aggregate the IOIMC models of the three modules of the CAS case study. This aggregation was performed using the two properties explained above. We found the exact same unreliability using standard compositional analysis and using the compositional analysis with extra aggregation. However, further work needs to be done to formally define this weaker equivalence.

Using extra aggregation we reduced the IOIMC models of the CAS modules to three IOIMC, each with only 6 states. Figure 6.9 shows these three models.

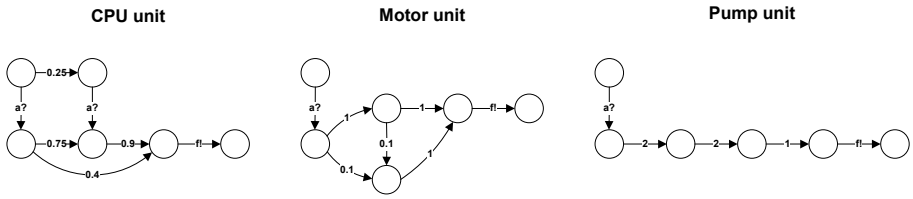

Figure 6.9: IOIMC models of the different units.

Compositional aggregation (using the weaker bisimulation) of these three models and the IOIMC model of the top OR-gate now resulted in a largest IOIMC of only 36 states and 119 transitions. Figure 6.10 shows a chart of the state space sizes of the IOIMC generated during compositional aggregation using the weaker equivalence.

6.2.2 Results

Table 6.3 shows the results of the CAS case study. For each analysis method the number of states and transitions is given for the largest appearing model (Markov chain or IOIMC).

We can see that the DIFTree approach is the most efficient in terms of state space. In this approach the three modules of the CAS DFT are translated to Markov chains, the largest (the

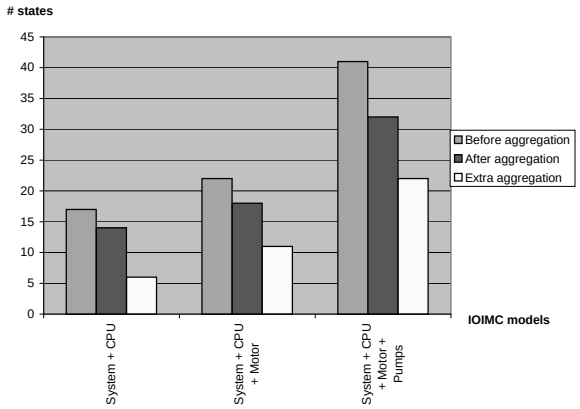

Figure 6.10: Chart of the compositional analysis of the CAS using the weaker bisimulation.

Approach	Max. Statespace	Transitions	Unreliability
Pure MC generation	85	526	0.6579
DIFTree	8	10	0.6579
Compositional	808	3660	0.6579
Extra aggregation	36	119	0.6579

Table 6.3: Results for the CAS case study.

pump module) having 8 states. These Markov chains are then used to find the unreliability of each module and these figures are then used to calculate the unreliability of the cardiac assist system using very efficient BDD techniques. See also Section 2.4 for more information on the DIFTree approach to analyzing DFT.

If we do not employ the modularizing techniques of the DIFTree approach and simply generate a Markov chain representing the entire DFT it is more than 10 times as large which shows the effectiveness of the DIFTree approach. However this Markov chain with 85 states is still a lot smaller than the largest IOIMC which is generated during compositional analysis of the DFT. The compositional analysis can in turn be improved by using a weaker bisimulation for the aggregation of IOIMC as explained in the previous subsection.

We can conclude that the modularization technique employed in the DIFTree approach is very effective for DFT with distinct modules. It should be noted though that this technique is only available if the top node of the DFT is static. For instance if the top node of the CAS DFT were a priority-AND gate then the modularization would not be applicable as we have also seen in the HCPS case study.

We can also conclude that the equivalence relation used to aggregate IOIMC has a huge impact on the efficiency of compositional analysis of DFT. Especially the concept of disregarding ‘uninteresting’ behavior (see Subsection 8.3.1) has a major impact on the reduction achieved by aggregating IOIMC.

Finally the CAS case study shows that compositional analysis does not always perform better (i.e. generate smaller models) than the traditional approach to analyzing DFT. In fact,

further research has shown examples of DFT which can much more efficiently be analyzed using the DIFTree approach than using compositional analysis. The DIFTree approach is particularly effective in cases where it can employ its modularization techniques and we have also seen that compositional analysis performs poorly when dealing with highly interconnected DFT, although by using a weaker equivalence this problem should be alleviated somewhat.

6.3 Multi-processor distributed computer system

The last case study is based on a multi-processor distributed computer system (MDCS) introduced in [22]. To this real-life fault-tolerant computer system we have added some dynamic elements to make it more realistic and complex in the same vein as in [25]. Figure 6.11 shows the DFT of the MDCS.

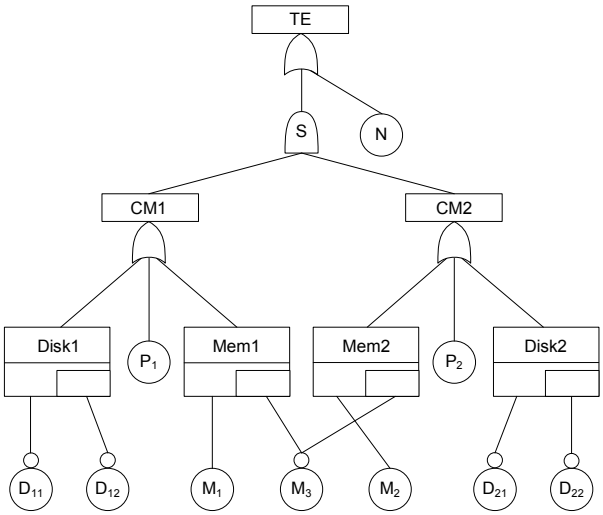

Figure 6.11: DFT for the MDCS case study.

The MDCS consists of a bus (N), two processors (P_1 and P_2), four hard disks (D_{11}, D_{12}, D_{21} and D_{22}) and three memories (M_1, M_2 and M_3). The processors, disks and memories are divided between two computing modules ('CM1' and 'CM2'). The failure rates for the basic events can be found in table 6.4.

This DFT was analyzed using the compositional aggregation approach and the results of this analysis are given in the following subsection.

6.3.1 Results

For the MDCS case study we only give the results for the compositional analysis with extra aggregation. Using only Tipp tool's aggregation leads to state spaces in excess of 1000 states. Table 6.5 gives the results for this case study.

For this case study we see that compositional analysis leads to a smaller largest IOIMC model than the CTMC obtained with the DIFtree method. This result seems to indicate that for

Component	Active failure rate	Dormant failure rate
N	2	0
P_1,P_2	500	0
D_{11},D_{21}	80000	40000
D_{12},D_{22}	80000	40000
M_1,M_2	30	0
M_3	30	15

Table 6.4: Failure rates for the MDCS case study in failures per 10^9 hours

Approach	Max. Statespace	Transitions	Unreliability
DIFtree	253	1383	$2.00025 \cdot 10^{-9}$
Extra aggregation	157	756	$2.00025 \cdot 10^{-9}$

Table 6.5: Results for the MDCS case study.

many DFT the compositional aggregation approach is more efficient than the DIFtree approach. The fact that Tipp tool's weak Markovian bisimulation leads to relatively large state spaces underlines the fact that it is highly desirable to implement tools supporting weak bisimulation for IOIMC.

Chapter 7

Tool support

Tool support for the research described in this thesis is provided by two tools: the DFT2Tipp tool created specifically for this research and the TippTool [16].

The DFT2Tipp tool translates a DFT specification to a TippTool specification. This translation is performed in 3 steps:

- Parsing of the DFT specification. See Section 7.2.
- Linking of the DFT elements parsed from the DFT specification. See Section 7.3.
- Writing the TippTool specification. See Section 7.4.

Before describing the process of translation the usage of the DFT2Tipp tool is explained in Section 7.1. Lastly we also describe how DFT2Tipp ties in with TippTool to analyze Dynamic Fault Trees in Section 7.5.

Throughout this section we will use a running example to show the inner workings of the DFT2Tipp tool and how it works together with the TippTool. The DFT specification for this example is shown below:

```
toplevel "A";
"A" and "B" "C"; "B" lambda=4.00 dorm=0.00; "C" lambda=2.00
dorm=1.00;
```

This example DFT is shown in Figure 7.1.

7.1 Usage

The DFT2Tipp tool must be run using a UNIX operating system. The tool reads from standard input and writes to standard output. This means that by simply calling dft2tipp you can type the DFT specification into the console finishing with an EOF character (Ctrl-D on most UNIX systems). The TippTool specification is then printed to the same console. Of course it is more common to read the DFT specification from a file and to write the TippTool specification to a file. This can be accomplished by piping the input and output. The following command reads the DFT specification from file spec1.dft and writes the TippTool specification to spec1.tpp:

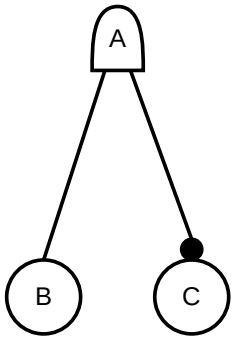

Figure 7.1: Example of a DFT

```
dft2tipp <spec1.dft >spec1.tpp
```

The format of the DFT specifications is that of the tool Galileo. The format of the output is the where-section of a TippTool specification. For more details we refer to the manuals of Galileo [2] and TippTool [16].

7.2 Parsing

The parsing of the DFT specification is done using the standard UNIX programming tools lex & yacc [20]. Lex was used to generate a lexical analyzer and yacc was used to generate a parser for DFT specifications. We refer to the files dft.l and dft.y and the lex&yacc documentation (available in any Unix operating system's man pages) for more info. The result of parsing is a list of unlinked DFT elements. During parsing the following fields are filled in for each DFT element:

- type: The type of the DFT element. Either gate or basic event.
- name: The name of the DFT element.
- line: The line of the DFT specification on which the DFT element is defined.
- for DFT gates:
  1. g_type: The type of the gate.
  2. inputs: A list of the names of the inputs of the gate.
  3. threshold (for voting gates only): The threshold of the voting gate.
  4. n_inputs (for voting gates only): The total number of non-trivial inputs of the voting gate.
- for basic events:
  1. distr: The distribution of the basic event.
  2. cov: The coverage attribute of the basic event.

3. res: The restoration attribute of the basic event.
4. dorm: The dormancy attribute of the basic event.
5. repl: The replication attribute of the basic event.
6. phase: The phase of the basic event.

After parsing, our running example results in three DFTElements, which are shown in Figure 7.2:

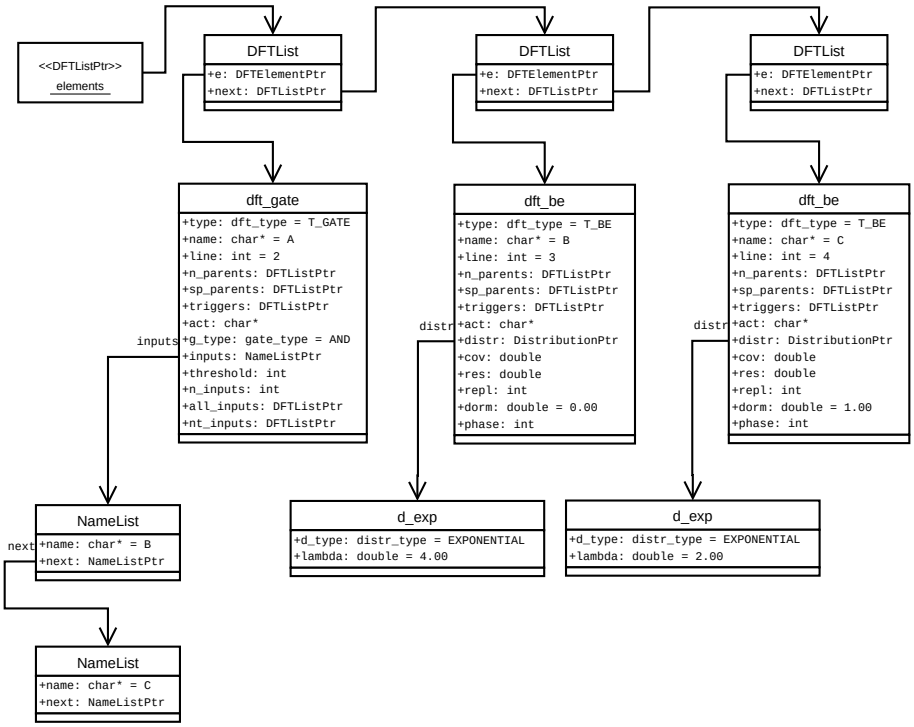

Figure 7.2: Result of parsing

The other fields of the DFT element are set during linking which is discussed in the next subsection.

7.3 Linking

During linking the DFT elements are connected in the following ways:

- Gates have a list of pointers to their children. (only for gates)
- Children have a list of pointers to their parents.
- Spares have a list of pointers to their sharing spare gates.
- Dependent elements have a list of pointers to their triggers.

The function check_and_link() also checks the integrity of the DFT specification (such as whether or not the specified inputs of a gate are present in the DFT specification). The function check_parents() is then called to check that there are no disconnected DFT elements. Finally the function set_activation() uses the new links to determine for each DFT element what its activation signal is.

After linking, our running example still consists of three DFTElements, which are now linked together. These DFTElements can be seen in Figure 7.3.

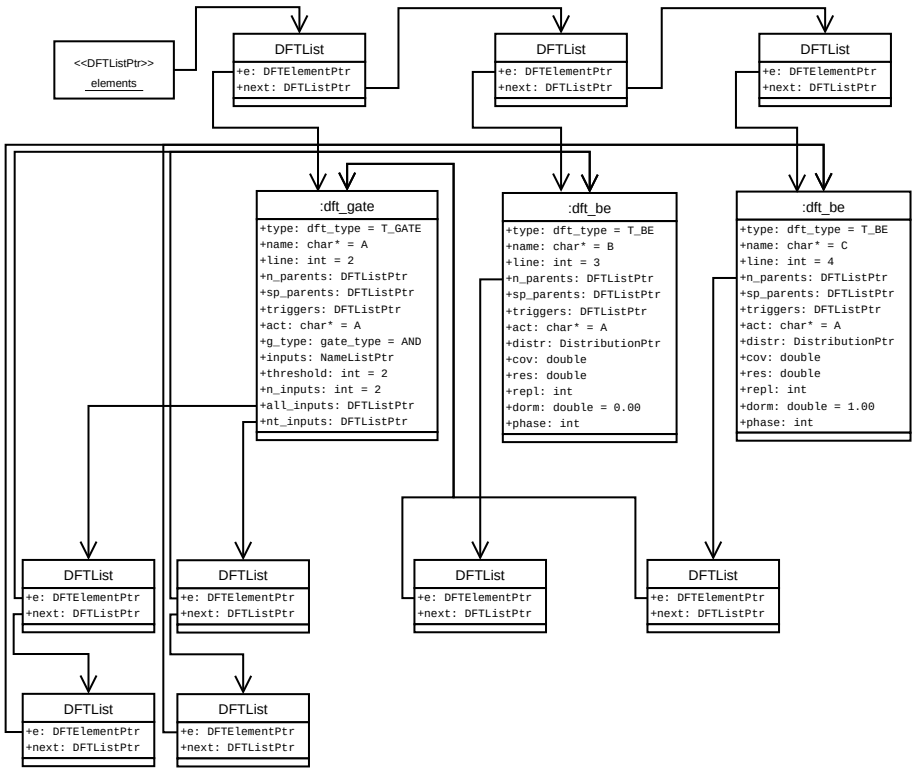

Figure 7.3: Result of linking

Basic events "B" and "C" are in gate "A"'s list of all inputs and its list of non-trivial inputs. The other way around "A" is in "B" and "C"'s list of normal parents. We also see that the function set_activation() has set the DFTElement::act fields for all three DFT elements. Because there are no spare gates in this DFT all elements have the same activation-signal, namely that of the top element of the DFT.

7.4 Output

The function translate() uses the list of linked DFT elements generated from the DFT specification to write a partial TippTool specification to standard output. This output describes a number of Input/Output Interactive Markov Chains, which together describe the behavior of the DFT. The output is divided into three parts: the action-signatures of the IOIMC, the def-

initions of the IOIMC and the definitions of the necessary TippTool processes. The result of running dft2tipp on our example is given below:

```
(*%%

A(f_B,f_C)(f_A);

B(a_A)(f_B);

C(a_A)(f_C);


  %%*)

process A :=
        GATE_2_2_0[f_B,f_C,f_A] endproc

process B :=
        BE_PASSIVE_COLD[a_A,f_B](4.000000) endproc

process C :=
        BE_PASSIVE_HOT[a_A,f_C](2.000000) endproc


(*%%*)

process GATE_2_2_0[input_1,input_2,out] :=
        input_1;GATE_2_2_1[input_2,input_1,out] []
        input_2;GATE_2_2_1[input_1,input_2,out] endproc

process BE_PASSIVE_COLD[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) endproc

process BE_PASSIVE_HOT[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process GATE_2_2_1[input_1,red_2,out] :=
        input_1;GATE_2_2_2[red_2,input_1,out] []
        red_2;GATE_2_2_1[input_1,red_2,out] endproc

process BE_ACTIVE[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc
```

```
process BE_FIRING[act,out] :=
        act;BE_FIRING[act,out] []
        out;BE_DONE[act] endproc

process GATE_2_2_2[red_1,red_2,out] :=
        red_1;GATE_2_2_2[red_1,red_2,out] []
        red_2;GATE_2_2_2[red_1,red_2,out] []
        out;GATE_DONE_2[red_1,red_2] endproc

process BE_DONE[act] :=
        act;BE_DONE[act] endproc

process GATE_DONE_2[red_1,red_2] :=
        red_1;GATE_DONE_2[red_1,red_2] []
        red_2;GATE_DONE_2[red_1,red_2] endproc


(*%%*)
```

The three parts of the output are separated by %%'s (note that (∗ and ∗) are used by TippTool to denote comments, so the first part of the output is in fact commented as are the separating %%'s). The three parts are described individually in the following subsections.

7.4.1 Action signatures

The function output_sigs() writes the action signatures of the IOIMC to the standard output. These action signatures list the input- and output-actions for each IOIMC. The format for an action signature is:

```
name(input-actions)(output-actions)
```

This part of the output of the dft2tipp tool is written as a comment in the TippTool specification and is included to make it possible to properly analyze the behavior of the entire DFT using compositional aggregation. In other words, we need to know the action signatures of the IOIMC to figure out on which actions we have to synchronize and when we can hide certain actions. Note that the compositional Markov chains used by the TippTool do not have action-signatures themselves.

For our running example we get the following action signatures:

```
A(f_B,f_C)(f_A);

B(a_A)(f_B);

C(a_A)(f_C);
```

We can see, for instance that the AND-gate "A" has input-actions "f_B" and "f_C", which correspond to the firing of basic events "B" and "C". Its only output-action is "f_A" which corresponds with the firing of the AND-gate itself. We can also see from these action signatures that the IOIMC community (representing the behavior of the entire DFT) has one input signal "a_A" and one output signal "f_A" representing the activation and firing of the DFT respectively.

7.4.2 IOIMC definitions.

All the IOIMC behave like one of the TippTool processes provided by the DFT2Tipp tool. These processes are generic and have variable actions and delay-rates. These variables are filled in for each IOIMC definition. The IOIMC definitions are printed to the standard output by the function output_defs(). Our running example gives us the following three IOIMC definitions:

```
process A :=
        GATE_2_2_0[f_B,f_C,f_A] endproc

process B :=
        BE_PASSIVE_COLD[a_A,f_B](4.000000) endproc

process C :=
        BE_PASSIVE_HOT[a_A,f_C](2.000000) endproc
```

We can see in this example that IOIMC A (which corresponds to the AND-gate A in the DFT) is defined as a TippTool process which behaves exactly like the process GATE_2_2_0 with actions "f_B", "f_C" and "f_A". The process name GATE_2_2_0 should be interpreted as follows:

- GATE denotes that this is a process that describes a standard gate.
- The first "2" denotes that the gate has 2 input signals.
- The second "2" denotes that the gate has a threshold of 2 (this means that the gate fires when 2 of its inputs have fired).
- The final "0" denotes that so far none of the gate's inputs have fired.

Of course the three processes used in these IOIMC definitions - GATE_2_2_0, BE_PASSIVE_COLD and BE_PASSIVE_HOT - must be defined somewhere. During the printing of the IOIMC definitions the tool keeps track of the processes that need to be defined in the global variable procs. At the end of the printing of the IOIMC definitions of our running example procs contains the three processes, which are shown in Figure 7.4.

7.4.3 Process definitions

The function output_procs() passes through all the processes in a process-list printing their definitions to the standard output. An example of a process definition is given below:

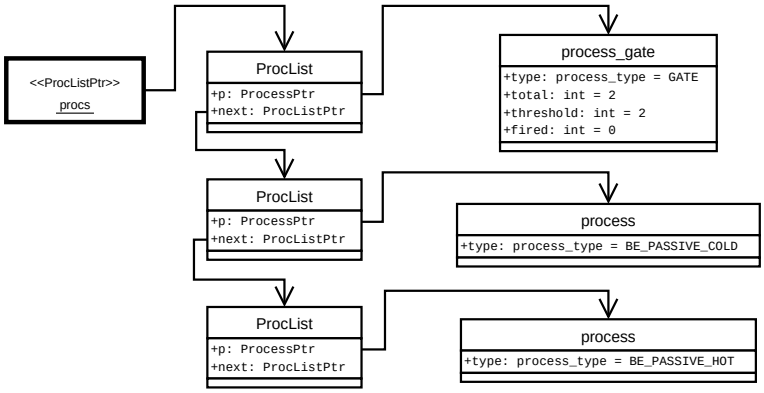

Figure 7.4: Process list after writing IOIMC definitions

```
process GATE_3_2_1[input_1,input_2,red_3,out] :=
        input_1;GATE_3_2_2[input_2,red_3,input_1,out] []
        input_2;GATE_3_2_2[input_1,red_3,input_2,out] []
        red_3;GATE_3_2_1[input_1,input_2,red_3,out] endproc
```

The left side of the equation tells us that this is the definition of process GATE_3_2_1, which has four variable actions: "input_1", "input_2", "red_3" and "out". On the right side of the equation we see a number of possible actions followed by the process these actions result in. These different possibilities are separated by the choice-operator: "[ ]". A schematic representation of this process definition is given in Figure 7.5.

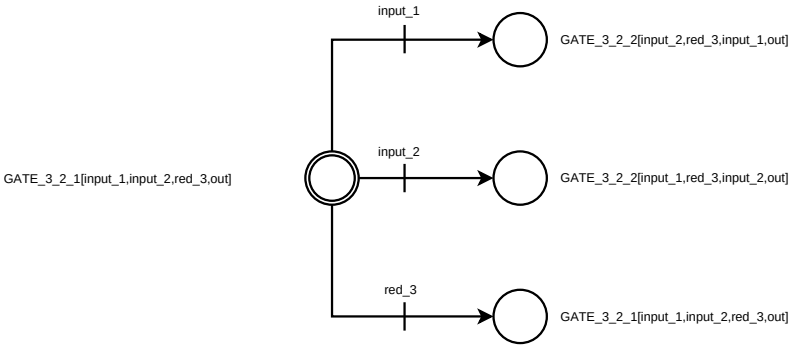

Figure 7.5: Schematic of a process definition

Initially only the processes found on the right side of the IOIMC definitions are written by the function output_procs(). This function, however, also adds the processes that appear on the right side of process definitions to the process-list. To make sure that processes are not defined twice the function add_Proc() checks whether the process being added is already present in the process-list. The nature of the processes ensures that the function output_procs() finishes at some point. Our running example gives us the following process definitions (Note that the first three processes correspond to the processes found on the right side of the IOIMC definitions):

```
process GATE_2_2_0[input_1,input_2,out] :=
        input_1;GATE_2_2_1[input_2,input_1,out] []
        input_2;GATE_2_2_1[input_1,input_2,out] endproc

process BE_PASSIVE_COLD[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) endproc

process BE_PASSIVE_HOT[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process GATE_2_2_1[input_1,red_2,out] :=
        input_1;GATE_2_2_2[red_2,input_1,out] []
        red_2;GATE_2_2_1[input_1,red_2,out] endproc

process BE_ACTIVE[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process BE_FIRING[act,out] :=
        act;BE_FIRING[act,out] []
        out;BE_DONE[act] endproc

process GATE_2_2_2[red_1,red_2,out] :=
        red_1;GATE_2_2_2[red_1,red_2,out] []
        red_2;GATE_2_2_2[red_1,red_2,out] []
        out;GATE_DONE_2[red_1,red_2] endproc

process BE_DONE[act] :=
        act;BE_DONE[act] endproc

process GATE_DONE_2[red_1,red_2] :=
        red_1;GATE_DONE_2[red_1,red_2] []
        red_2;GATE_DONE_2[red_1,red_2] endproc
```

7.5 TippTool

The result of the DFT2Tipp tool can be used to analyze the original DFT using TippTool. In this section we will describe how this analysis can be performed. First we will see how the IOIMC community generated by DFT2Tipp can be combined to form a single, weakly bisimilar, IOIMC. Consequently we will look at how we can use TippTool to analyze this IOIMC to calculate interesting fault tolerance measures for the Dynamic Fault Tree. Lastly we will look at two possible ways to improve the aggregation of IOIMC.

7.5.1 Compositional Aggregation

A TippTool specification takes on the following form [16]:

```
specification NAME

behaviour

        BEHAVIOUR FORMULA

where

        PROCESS DEFINITIONS

endspec
```

The output of the DFT2Tipp tool are process definitions which can be used as such in a TippTool specification. The name of this specification can be chosen arbitrarily. In this subsection we describe the process of compositional aggregation that can be used to compose and aggregate IOIMC models. Before composing two IOIMC models it is recommended that these models are first aggregated. If we look at IOIMC A in our running example, for instance we see that before aggregating it the IOIMC has 7 states (see Figure 7.6).

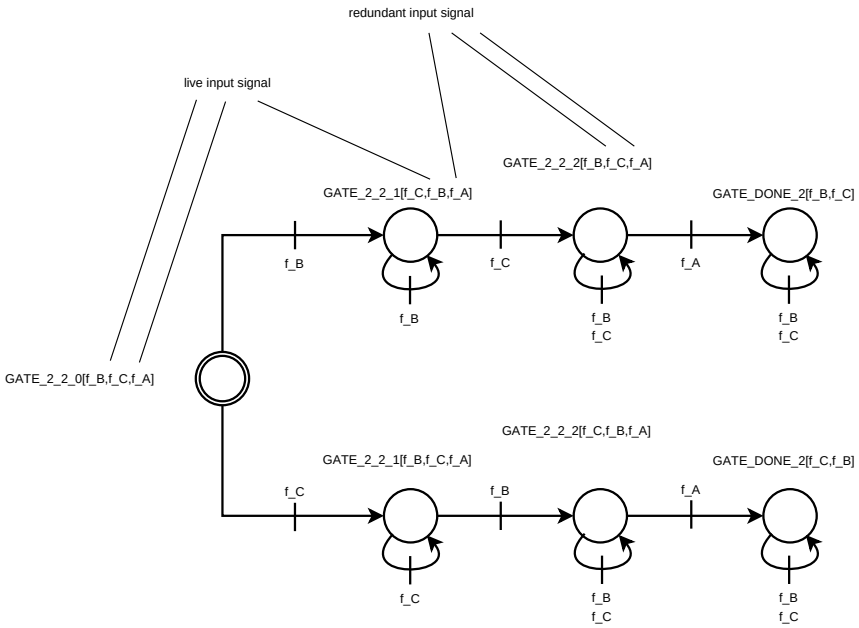

Figure 7.6: Schematic of IOIMC A as interpreted by TippTool

It is obvious that states GATE_DONE_2[f_B,f_C] and GATE_DONE_2[f_C,f_B] are equivalent except for the ordering of their actions. In the same way states GATE_2_2_2[f_B,f_C,f_A] and GATE_2_2_2[f_C,f_B,f_A] are equivalent. The reason TippTool sees these states as

different is that their actions are ordered differently. Below we show how to aggregate a single process with respect to weak bisimilarity using TippTool.

IOIMC A is defined in the process definitions generated by DFT2Tipp as process A. To aggregate this process we first specify it as the behavior. The name of the specification is chosen arbitrarily and the process definitions generated by DFT2Tipp are placed in the where-section:

```
specification compaggr

behaviour

        A

where

(*%%

A(f_B,f_C)(f_A);

B(a_A)(f_B);

C(a_A)(f_C);


  %%*)

process A :=
        GATE_2_2_0[f_B,f_C,f_A] endproc

process B :=
        BE_PASSIVE_COLD[a_A,f_B](4.000000) endproc

process C :=
        BE_PASSIVE_HOT[a_A,f_C](2.000000) endproc


(*%%*)

process GATE_2_2_0[input_1,input_2,out] :=
        input_1;GATE_2_2_1[input_2,input_1,out] []
        input_2;GATE_2_2_1[input_1,input_2,out] endproc

process BE_PASSIVE_COLD[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) endproc
```

```
process BE_PASSIVE_HOT[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process GATE_2_2_1[input_1,red_2,out] :=
        input_1;GATE_2_2_2[red_2,input_1,out] []
        red_2;GATE_2_2_1[input_1,red_2,out] endproc

process BE_ACTIVE[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process BE_FIRING[act,out] :=
        act;BE_FIRING[act,out] []
        out;BE_DONE[act] endproc

process GATE_2_2_2[red_1,red_2,out] :=
        red_1;GATE_2_2_2[red_1,red_2,out] []
        red_2;GATE_2_2_2[red_1,red_2,out] []
        out;GATE_DONE_2[red_1,red_2] endproc

process BE_DONE[act] :=
        act;BE_DONE[act] endproc

process GATE_DONE_2[red_1,red_2] :=
        red_1;GATE_DONE_2[red_1,red_2] []
        red_2;GATE_DONE_2[red_1,red_2] endproc


(*%%*)


endspec
```

To aggregate this specification with respect to weak bisimilarity we call TippTool as follows:

```
tipptool -i compaggr_A
```

In this example our TippTool specification is saved in the file compaggr_A.tpp. The flag '-i' denotes that TippTool should aggregate using IM (Interactive Markovian) bisimulation. The resulting TippTool specification (written to compaggr_Ared.tpp) is given below:

```
specification compaggr_Ared
behaviour
```

```
   VQ1

where

   process VQ1 := f_C; VQ2 [] f_B; VQ3 endproc
   process VQ2 := f_C; VQ2 [] f_B; VQ4 endproc
   process VQ3 := f_B; VQ3 [] f_C; VQ4 endproc
   process VQ4 := f_B; VQ4 [] f_C; VQ4 [] f_A; VQ5 endproc
   process VQ5 := f_B; VQ5 [] f_C; VQ5 endproc

endspec
```

By adding processes VQ$*$ to the original specification and changing the IOIMC definition of A we can replace IOIMC A's original behavior with the aggregated behavior:

```
  process A := VQ1 endproc

  process VQ1 := f_C; VQ2 [] f_B; VQ3 endproc
  process VQ2 := f_C; VQ2 [] f_B; VQ4 endproc
  process VQ3 := f_B; VQ3 [] f_C; VQ4 endproc
  process VQ4 := f_B; VQ4 [] f_C; VQ4 [] f_A; VQ5 endproc
  process VQ5 := f_B; VQ5 [] f_C; VQ5 endproc
```

After reducing all the IOIMC in this way we can start the compositional aggregation of the IOIMC community. Compositional aggregation consists of consecutively composing and aggregating two IOIMC until only one IOIMC is left. To specify that we want to analyze the composition of two processes the behavior clause of the TippTool specification must be altered. Below, for instance, we see the behavior clause for the analysis of the composition of the IOIMC A and B:

```
 hide f_B in (A |[f_B]| B)
```

IOIMC A and B are synchronized on action 'f_B' (the only action the two IOIMC share). This action is then hidden since no other IOIMC in the community has the action in its action signature. The resulting aggregated IOIMC is given below:

```
specification compaggr_ABred
behaviour

   YW1

where

   process YW1 := f_C; YW5 [] a_A; YW6 endproc
   process YW2 := a_A; YW2 [] f_C; YW3 endproc
```

```
   process YW3 := f_C; YW3 [] a_A; YW3 [] f_A; YW4 endproc
   process YW4 := f_C; YW4 [] a_A; YW4 endproc
   process YW5 := f_C; YW5 [] a_A; YW7 endproc
   process YW6 := (tau, 4.0); YW2 [] a_A; YW6 [] f_C; YW7 endproc
   process YW7 := (tau, 4.0); YW3 [] f_C; YW7 [] a_A; YW7 endproc

endspec
```

We now update the original specification, replacing the IOIMC definitions (and action signatures) of IOIMC A and B with one IOIMC definition (and action signature) A_B (note that the behavior-clause is set to compose and aggregate IOIMC A_B and C):

```
specification compaggr

behaviour

        hide f_C in (A_B |[f_C,a_A]| C)

where

(*%%

A_B(f_C,a_A)(f_A)

C(a_A)(f_C);


  %%*)

process A_B :=
        YW1 endproc

process C :=
        BE_PASSIVE_HOT[a_A,f_C](2.000000) endproc


(*%%*)

process GATE_2_2_0[input_1,input_2,out] :=
        input_1;GATE_2_2_1[input_2,input_1,out] []
        input_2;GATE_2_2_1[input_1,input_2,out] endproc

process BE_PASSIVE_COLD[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) endproc
```

```
process BE_PASSIVE_HOT[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process GATE_2_2_1[input_1,red_2,out] :=
        input_1;GATE_2_2_2[red_2,input_1,out] []
        red_2;GATE_2_2_1[input_1,red_2,out] endproc

process BE_ACTIVE[act,out](rate) :=
        act;BE_ACTIVE[act,out](rate) []
        (tau,rate);BE_FIRING[act,out] endproc

process BE_FIRING[act,out] :=
        act;BE_FIRING[act,out] []
        out;BE_DONE[act] endproc

process GATE_2_2_2[red_1,red_2,out] :=
        red_1;GATE_2_2_2[red_1,red_2,out] []
        red_2;GATE_2_2_2[red_1,red_2,out] []
        out;GATE_DONE_2[red_1,red_2] endproc

process BE_DONE[act] :=
        act;BE_DONE[act] endproc

process GATE_DONE_2[red_1,red_2] :=
        red_1;GATE_DONE_2[red_1,red_2] []
        red_2;GATE_DONE_2[red_1,red_2] endproc


(*%%*)

  process YW1 := f_C; YW5 [] a_A; YW6 endproc
  process YW2 := a_A; YW2 [] f_C; YW3 endproc
  process YW3 := f_C; YW3 [] a_A; YW3 [] f_A; YW4 endproc
  process YW4 := f_C; YW4 [] a_A; YW4 endproc
  process YW5 := f_C; YW5 [] a_A; YW7 endproc
  process YW6 := (tau,4.0); YW2 [] a_A; YW6 [] f_C; YW7 endproc
  process YW7 := (tau,4.0); YW3 [] f_C; YW7 [] a_A; YW7 endproc

endspec
```

After the composition and aggregation of IOIMC A_B and C we are left with a single IOIMC and so compositional aggregation is finished. In the next subsection we will see how we can analyze this IOIMC to determine the Dynamic Fault Tree's unreliability.

7.5.2 Analysis using TippTool

In this subsection we explain how the IOIMC representing the behavior of a DFT can be analyzed. Let's take a look at the resulting IOIMC of our running example:

```
specification compaggr_ABCred
behaviour

   TT1

where

   process TT1 := (tau, 2.0); TT5 [] a_A; TT6 endproc
   process TT2 := a_A; TT2 endproc
   process TT3 := f_A; TT2 [] a_A; TT3 endproc
   process TT4 := (tau, 4.0); TT3 [] a_A; TT4 endproc
   process TT5 := a_A; TT4 endproc
   process TT6 := (tau, 2.0); TT4 [] a_A; TT6 [] (tau, 4.0); TT7 endproc
   process TT7 := (tau, 2.0); TT3 [] a_A; TT7 endproc

endspec
```

We can see that this IOIMC has two actions: 'a_A' and 'f_A' (see Subsection 7.4.1). In fact when transforming Dynamic Fault Trees into IOIMC using DFT2Tipp and compositional aggregation the resulting IOIMC always have two actions: an activation action and a firing action. These actions are always named 'a_top' and 'f_top' respectively, where 'top' is the toplevel element of the DFT. Action 'a_top' is an input-action which represents the activation of the DFT and 'f_top' is an output-action, which represents the firing of the DFT. In other words these actions represent the switching on and breaking down of the system under analysis.

The unreliability of a system is the probability that the system fails within a certain time period. Looking at our example IOIMC this is the probability that the TippTool process has executed action 'f_A' within that time period. For continuous-time Markov chains the probability that the CTMC is in a certain state after a certain period of time can be calculated using differential equations. The presence of actions in TippTool processes however makes it unclear how to calculate such measures. This issue is handled in TippTool by assuming that all actions take place immediately. This interpretation is very convenient to us as this means that the IOIMC representing DFT are interpreted to activate immediately and also to fire immediately. This is exactly the behavior we want for our IOIMC: there should be no dormant period when the system starts operating and there should be no delay in the failing of the system when the right conditions are met.

TippTool can compute the probability that a process is in a certain state at a certain time. In our example we want to know the probability that process TT1 is in state TT2 after a certain period of time, since TT2 is the state in which the system has failed. To calculate this measure we must add a measure definitions file in the same directory as the TippTool definition. For our example this file looks as follows:

```
STATEMEASURE Failed TT2
```

The file must have the same name as the TippTool specification only with extension .mdf. Now we run the following command to calculate the measure with a time period of 4 time units:

```
tipp -s jensen -t 4 compaggr_ABCred
```

The result of this calculation (in this case 0.99966253) is written both to standard output and to the file compaggr_ABCred.sm.Failed, since we named the measure 'Failed'.

For our example calculating the unreliability was quite easy but for other DFT it will be more complicated:

- In some cases the IOIMC will still express behavior after the system has failed. In this case we must calculate the probability that the process is in a set of states instead of a single state. The difficulty will be in determining in which states the system has failed.
- When Priority-AND gates are used the system may enter a state in which it can no longer fail (it is up to the designer of the DFT to decide whether this is desired behavior). Unfortunately using DFT2Tipp and TippTool to transform such a DFT into an IOIMC will result in an IOIMC that doesn't differentiate between states in which the system has failed and states in which the system can no longer fail.
- There are Dynamic Fault Trees that express non-deterministic behavior. This is usually caused by multiple events happening at the same time due to a shared dependency or shared input. TippTool handles non-determinism by assuming that all possible outcomes have the same probability of occurring. This is not desired behavior, since the distribution of a non-deterministic choice is unknown instead of equiprobable.

The first two problems shown above can be solved by parallel composing the IOIMC under analysis with a so-called Observer IOIMC. This IOIMC activates the DFT and then responds to the firing of the DFT by moving to a failed state. Below we see the TippTool definition of an Observer IOIMC:

```
process Observer := a_A;Activated [] f_A;Failed endproc
process Activated := f_A; Failed endproc
process Failed := f_A; Failed endproc
```

We now parallel compose this Observer IOIMC with the IOIMC of the DFT under analysis and hide the activation and firing signal. The resulting TippTool specification for our running example is given below:

```
specification compaggr_ABCred
behaviour

   hide a_A,f_A in TT1 |[a_A,f_A]| Observer
```

```
where

   process TT1 := (tau, 2.0); TT5 [] a_A; TT6 endproc
   process TT2 := a_A; TT2 endproc
   process TT3 := f_A; TT2 [] a_A; TT3 endproc
   process TT4 := (tau, 4.0); TT3 [] a_A; TT4 endproc
   process TT5 := a_A; TT4 endproc
   process TT6 := (tau, 2.0); TT4 [] a_A; TT6 [] (tau, 4.0); TT7 endproc
   process TT7 := (tau, 2.0); TT3 [] a_A; TT7 endproc

   process Observer := a_A;Activated [] f_A;Failed endproc
   process Activated := f_A; Failed endproc
   process Failed := f_A; Failed endproc
endspec
```

The states of the composite IOIMC will look like 'TT1||Observer' or 'TT2||Failed'. We can now change our measure definitions file to calculate the probability that the IOIMC is in any state containing the string 'Failed':

```
STATEMEASURE Failed Failed
```

The unreliability can now be found once again by running 'tipp'. This approach will also work for IOIMC with more than one failed state or IOIMC which do not differentiate between failed and non-failed states.

Chapter 8

Conclusion

In this chapter we will draw conclusions from the research we have done on the compositional analysis of DFT using IOIMC. We will compare the results of the research with the goals set out in Section 1.2 and we will also look forward to possible avenues of research that could be pursued in the future.

8.1 Formalizing dynamic fault trees

A formal syntax for DFT was realized by defining a DFT as a directed acyclic graph and then by imposing a number of restrictions on this graph (see Subsection 4.1). This formalization has removed some of the vagueness of the DFT formalism. We can now strictly decide whether a DFT is valid by checking if the restrictions of Definition 4.2 hold. The formal semantics of DFT was defined by specifying an IOIMC model for each DFT element and by defining how these IOIMC models could be combined to define the behavior of an entire DFT (see Chapter 4). This has also helped remove the former vagueness of the DFT formalism. In particular the problems regarding simultaneity, the occurrence of several events simultaneously, where solved by using non-determinism to model the fact that we do not always know the exact order in which events affect a system.

We can thus conclude that the formalization of the DFT formalism was successful. It should also be noted that we have extended the DFT formalism by allowing more complex spares and dependencies (see Section 4.2).

Another advantage of the compositional approach to defining DFT semantics it that the DFT formalism can now be extended or altered more easily. An alteration to one of the DFT gates can be realized simply by changing that gate's IOIMC representation and this does not affect the other DFT elements. Similarly new gates with their own IOIMC representation can be added to the DFT formalism. Finally it is possible to extend the DFT formalism in total, for instance by adding a notion of repair [26]. This change could also be implemented by changing the IOIMC representations of the different DFT elements.

8.2 Compositional analysis

By finding an IOIMC model of each DFT element it became possible to analyze DFT in a compositional way. Instead of translating the entire DFT into a large Markov chain the components (elements) of the DFT are translated into IOIMC models. These IOIMC models are then composed to find a single IOIMC which represents the behavior of the DFT (see Chapter 5).

The advantage of this method is that compositional aggregation can be employed to avoid the generation of large IOIMC models. In compositional aggregation the IOIMC models are aggregated at each step of the composition process using some equivalence. Each time two IOIMC models are composed the resulting model is first replaced by a smaller, equivalent, model before composing it with another IOIMC model.

In a number of case studies (see Chapter 6) we have compared the results of compositional analysis with the results of tradition DFT analysis techniques (see Section 2.4). First of all we found that both analysis techniques yielded the same numerical results which suggests that the results found by using compositional analysis can be trusted. Secondly we found cases in which compositional analysis greatly outperformed traditional analysis. Especially in DFT which have a simple, dynamic, structure with few connections but many basic events compositional analysis performs relatively well. We also found cases, however, in which traditional analysis outperformed the composition method. Especially in DFT with few basic events but many gates and interconnections traditional analysis performs relatively well. The modularization techniques employed in the DIFTree [23] approach to DFT analysis also drastically improve its performance in analyzing certain DFT. Finally we have seen that the choice of equivalence used to aggregate IOIMC models is crucial in combatting the state space explosion problem. A small improvement in aggregation can, because of the repetitive nature of compositional analysis, yield a huge decrease in state space in the appearing IOIMC models.

8.3 Future Work

In this section we suggest a number of topics for further research. First of all we look into improving IOIMC aggregation. Secondly we discuss the subject of ordering strategies for the compositional aggregation of IOIMC communities. Finally we examine the possibilities of analyzing DFT more thoroughly using advanced logical formulas.

8.3.1 Equivalences on IOIMC

As already noted it is crucial to find a good equivalence on IOIMC models. If the equivalence is too loose than the aggregated IOIMC model no longer adequately models the behavior of the (partial) DFT. This may lead to the compositional analysis yielding incorrect results. If the equivalence is too strict, however, the aggregated IOIMC will be larger than necessary. This problem is compounded by the fact that in compositional analysis IOIMC models are composed frequently. We feel that the equivalence used in the compositional analysis described in this thesis, namely weak bisimulation, is too strict. Below we suggest a number of ways to

make this equivalence less strict and we have seen in one case study (see Section 6.2) that these improvements can have a huge effect on the state spaces of the generated IOIMC models:

- **Output focusing.** For any IOIMC model we are really only interested in its external behavior. An IOIMC model uses this external behavior to communicate with the outside world. In fact, because IOIMC must be input-enabled, the only interesting behavior is an IOIMC model's output behavior. This aspect of IOIMC models is reflected in the IOIMC model of an entire DFT: we are then interested in when the IOIMC model signals its failure using its one output action. We can use this notion to split an IOIMC's state space into interesting and uninteresting states. Uninteresting states are those states that can never lead to any output action. We can now make our equivalence looser by simply not putting any restrictions on uninteresting states. This will result in the aggregation of all uninteresting states into one single state, potentially reducing the state space of the IOIMC model greatly.

- **Ignoring Markovian self-loops** In [4, Definition 13] a weak bisimulation is defined which does not take into account Markovian transitions from states in an equivalence class to states in the same equivalence class. This makes the bisimulation looser than the Markovian bisimulation in [15, Definition 3.4]. This concept could also be applied to our definition of weak bisimulation for IOIMC.

8.3.2 Ordering Strategies

In each step of compositional analysis two IOIMC models from an IOIMC community are chosen to be composed and then aggregated. In our studies we have found that the order in which the IOIMC models are chosen for compositional aggregation has a huge influence on the state spaces of the generated IOIMC models. In this thesis this choice has been made by hand in an experimental fashion, but it is highly desirable to be able to choose this order automatically in the future. It is therefore very useful to further study what is the best, or at least a good, ordering strategy for the compositional aggregation of an IOIMC community.

A number of factors influence the size of the largest appearing IOIMC during the compositional aggregation of a community:

- **More synchronization means less states.** First of all we can predict the size of a composite IOIMC model. In the worst case the state space of the composite IOIMC will be the cross-product of the state spaces of the two IOIMC being composed. This figure can be reduced drastically by choosing two IOIMC models which have a lot of matching signals. The more the two IOIMC models have to be synchronized the smaller the composite IOIMC will be.

- **More hiding means better aggregation.** We use weak bisimulation to aggregate the appearing IOIMC models. It is therefore to choose the composition order in such a way that the appearing IOIMC models can be aggregated significantly. Weak bisimulation is based primarily on abstracting from internal transitions. It is therefore logical that models with a lot of internal transitions can be aggregated a lot. Internal transitions appear when actions are hidden and therefore it is useful to choose IOIMC models for

composition in such a way that many signals can be hidden. Applying this principal often means that independent modules within a DFT are composed first since we can then hide all the internal signals within this module.

- **4x4 > 6x2**. As noted before the state space of a composite IOIMC model is a subset of the cross-product of the state spaces of its component IOIMC models. This meant that, in terms of state space size, it is much better to construct one large IOIMC model and compose it iteratively with small IOIMC models to reach the final IOIMC model than to construct two medium sized IOIMC models and to compose these two models as the final step in compositional aggregation.

- **Using knowledge of DFT element models.** We can use some knowledge of the IOIMC models of DFT elements to predict the best order of composition. For instance, we know that composing an IOIMC model of some DFT element with its auxiliary models will result in an IOIMC model with the same number of states, but more transitions than the original.

8.3.3 Advanced DFT analysis

In this thesis we have focused mainly on determining the unreliability of systems using DFT modelling. The unreliability of a system is the probability that it will fail within a certain time period. This is a much used fault-tolerance measure, but a very simple one. It seems reasonable to assume that there is also interest in more complex measures, for instance: the probability that some system component A fails before system component B or the probability that the system will fail while some component A is still operational.

Such complex properties can be expressed using advanced logics such as continuous stochastic logic (CSL) [1] used widely in model-checking [3]. CSL is applicable to *labelled CTMC* [3]. A labelled CTMC is a CTMC with a set of atomic propositions and a labelling function which assigns to each state the set of atomic propositions that hold true in that state.

So how does this apply to DFT analysis? A DFT is traditionally and in our approach analyzed by transforming it into a CTMC (without labelling). But if it were possible to translate a DFT into a labelled CTMC with an atomic propositions $Afailed$ for each DFT event A and with its states labelled to reflect the status of all the DFT events then we could check interesting CSL properties for this labelled CTMC. To transform a DFT into a labelled CTMC we could apply labels to IOIMC and follow the compositional analysis method described in this thesis. This extension of the IOIMC formalism which involves finding appropriate definitions for parallel composition, abstraction and aggregation lies beyond the scope of this thesis, but below we will outline some ideas for any future research in this area.

Let's consider the DFT in Figure 8.1. For the sake of simplicity we will ignore the notion of activation in this example. Basic event A has a failure rate of λ and B has a failure rate of μ. We want to transform this DFT into a labelled CTMC with three atomic propositions: *Afailed*, *Bfailed* and *Cfailed*. To do this we could transform the DFT into a community of three labelled IOIMC shown in Figure 8.2. We can now transform this labelled IOIMCC into a labelled CTMC using compositional aggregation. Under parallel composition sets of atomic

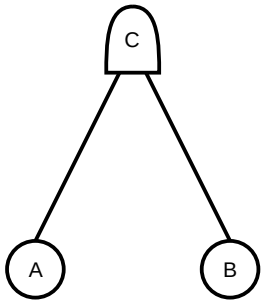

Figure 8.1: Example DFT.

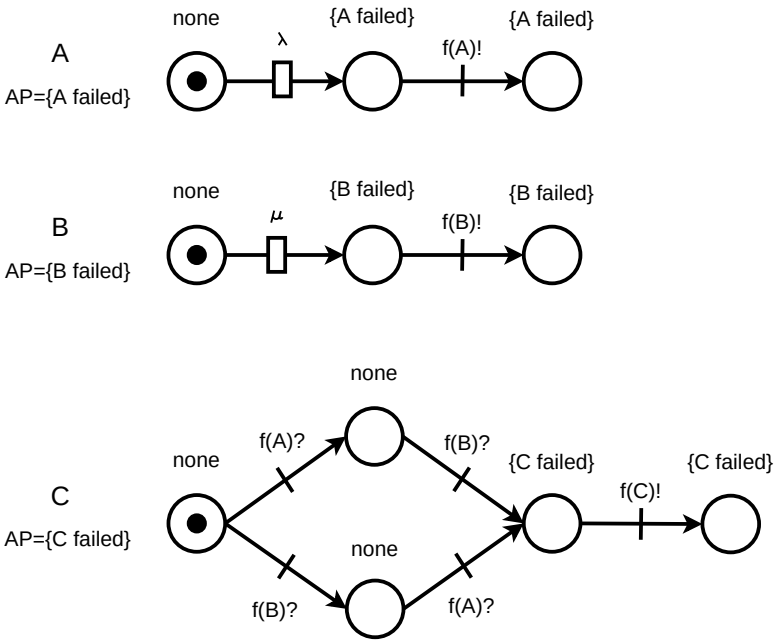

Figure 8.2: Community of labelled IOIMC which models the DFT in figure 8.1.

propositions are simply unified and we must ensure that the equivalence we use for aggregation preserves the labelling of the states. The resulting labelled CTMC is shown in Figure 8.3.

We could now use the labelled CTMC of Figure 8.3 to calculate such measures as, for instance, the probability that *A* will fail before *B* or the probability that *C* fails within a certain time period after *A* fails.

Incorporating the concept of dormancy (see Section 4.3) into this new approach should be quite simple. We could, for instance, for a basic event add an atomic proposition *Aactive* and assign it to all the active states of the IOIMC model of *A*. In that way we could also analyze properties involving the dormancy of the spares in a DFT.

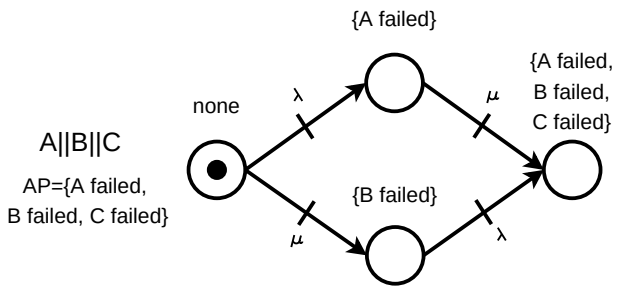

Figure 8.3: Labelled CTMC which models the DFT in figure 8.1.

Appendix A

Proofs

A.1 Theorem 3.3

For an IOIMC P relation $\approx_P$ we find:

1. *$\approx_P$ is the largest weak bisimulation on P.*

Proof. First we prove that for any IOIMC P $\approx_P$ is the largest weak bisimulation on P. To do this we must prove that $\approx_P$ is an equivalence relation by proving that it is reflexive, symmetric and transitive. The proofs for reflexivity and symmetry are quite simple, but the proof for transitivity is somewhat more complex.

For any I/O-IMC P it is trivial that the identity relation Id_P is a weak bisimulation. For any state s in S_P we find that $(s, s) \in \mathsf{Id}_P$ and thus $s \approx s$. In other words $\approx$ is reflexive. If $s \approx t$ then there is a weak bisimulation R with $(s, t) \in R$. Because R is an equivalence relation we also find that $(t, s) \in R$ and so $t \approx s$. We now find that $s \approx t$ implies $t \approx s$. This means that $\approx$ is symmetric.

Let R_1 and R_2 be two weak bisimulations on P and let $R_1{R_2}^*$ be defined as their *recursive composition*:

$$R_1{R_2}^* = \{(s,t) \mid \exists n \in \mathbb{N}, x_1, ..., x_n \in S^{all}.$$
$$n \geq 0 \wedge (s, x_1) \in R_1 \cup R_2 \wedge ... \wedge (x_n, t) \in R_1 \cup R_2\}$$

Because R_1 and R_2 are equivalence relations it is trivial that $R_1{R_2}^*$ is reflexive and symmetric. From the definition of $R_1{R_2}^*$ we can also immediately deduce that it is a transitive relation.

We will now prove that $R_1{R_2}^*$ is a weak bisimulation relation. Recall that if (s, t) is in $R_1{R_2}^*$ then there exist a series of states $x_1, ..., x_n$ such that they are stepwise related through either R_1 or R_2: $sR_ix_1R_ix_2...x_nR_it$ where R_i is R_1 or R_2. We find that for any action $a \in A_P^{all}$ with $s \overset{a}{\Longrightarrow} s'$ there is an x_1' with $x_1 \overset{a}{\Longrightarrow} x_1'$ and $(s', x_1') \in R_1 \cup R_2$. This in turn means that there is an x_2' with $x_2 \overset{a}{\Longrightarrow} x_2'$ and $(s', x_2') \in R_1 \cup R_2$. So we find a series of states $x_1', ..., x_n', t'$ such that $(s', x_1') \in R_1 \cup R_2 \wedge ... \wedge (x_n', t') \in R_1 \cup R_2$. Thus $t \overset{a}{\Longrightarrow} t'$ with $(s', t') \in R_1{R_2}^*$, which means that the first clause of Definition 3.11 holds. To prove that the second clause holds we must first consider the equivalence classes of $R_1{R_2}^*$.

From the definition we know that $R_1 \subseteq R_1{R_2}^*$ and $R_2 \subseteq R_1{R_2}^*$. This means that any equivalence class of $R_1{R_2}^*$ is exactly the union of one or more equivalence classes of R_1 and

it is also exactly the union of one or more equivalence classes of R_2. So for any equivalence class C of $R_1{R_2}^*$ we find that there is a set $C'_1, C'_2, ...C'_n$ of equivalence classes of R_1 and a set $C''_1, C''_2, ...C''_m$ of equivalence classes of R_2 with $n, m > 0$ such that:

$$C = C'_1 \cup C'_2 \cup ... \cup C'_n = C''_1 \cup C''_2 \cup ... \cup C''_m$$

It is trivial that this also holds for the internal backwards closure of an equivalence class of $R_1{R_2}^*$:

$$C^{int} = C'^{int}_1 \cup C'^{int}_2 \cup ... \cup C'^{int}_n = C''^{int}_1 \cup C''^{int}_2 \cup ... \cup C''^{int}_m$$

For a state s we find that it's equivalence class for $R_1{R_2}^*$: $[s]_{R_1{R_2}^*}$ contains all states t such that $(s,t) \in R_1{R_2}^*$. Because $(s,t) \in R_1$ implies $(s,t) \in R_1{R_2}^*$ we find that $[s]_{R_1} \subseteq [s]_{R_1{R_2}^*}$ and conversely $[s]_{R_2} \subseteq [s]_{R_1{R_2}^*}$. Because equivalence classes are disjunct we now find that if an equivalence class of $R_1{R_2}^*$: $C \neq [s]_{R_1{R_2}^*}$ then $C \cap [s]_{R_1} = \emptyset$ and also $C \cap [s]_{R_2} = \emptyset$.

Let $(s,t) \in R_1{R_2}^*$ and let there be a s' such that $s \Longrightarrow s'$ and s' is stable. If $(s,t) \in R_1 \cup R_2$ then the second clause of Definition 3.11 holds. Otherwise there exists an x_1 such that $(s,x_1) \in R_1 \cup R_2$, so either $(s,x_1) \in R_1$ or $(s,x_1) \in R_2$. For the first case we find that there is an x'_1 such that $x_1 \Longrightarrow x'_1$ and x'_1 stable and $\gamma_M(s', C'^{int}) = \gamma_M(x'_1, C'^{int})$, for all equivalence classes C' of R_1 except $[s']_{R_1}$. For the second case we find that there is an x'_1 such that: $x_1 \Longrightarrow x'_1$ and x'_1 stable and $\gamma_M(s', C'^{int}) = \gamma_M(x'_1, C''^{int})$, for all equivalence classes C'' of R_2 except $[s']_{R_2}$. So we find that there is always an x'_1 such that $x_1 \Longrightarrow x'_1$ and x'_1 stable and $\gamma_M(s', C^{int}) = \gamma_M(x'_1, C^{int})$, for all equivalence classes C of $R_1{R_2}^*$ except $[s']_{R_1{R_2}^*}$. Note that the equivalence classes $[s']_{R_1}$ and $[s']_{R_2}$ must be contained in $[s']_{R_1{R_2}^*}$ and not in any other equivalence class of $R_1{R_2}^*$. The same holds for the pairs (x_1, x_2), (x_2, x_3), ..., (x_n, t), so in the end we find that the second clause of Definition 3.11 holds for $R_1{R_2}^*$ so $R_1{R_2}^*$ is a weak bisimulation.

If we now have that $s \approx_P x$ and $x \approx_P t$ then there exist two weak bisimulations R_1 and R_2 such that $(s,x) \in R_1$ and $(x,t) \in R_2$. Then $(s,t) \in R_1{R_2}^*$ and because $R_1{R_2}^*$ is a weak bisimulation on P we find that $s \approx_P t$ which means that $\approx_P$ is an equivalence relation.

Because $\approx$ is the union of all weak bisimulations it is trivial that $\approx$ itself is a weak bisimulation now that we have proven that it is an equivalence relation. That $\approx$ is the largest weak bisimulation on P follows immediately from its definition.

A.2 Theorem 3.4

Weak bisimilarity is substitutive with parallel composition and hiding.

$$\begin{array}{lll} P_1 \approx P_2 & \textit{implies} & P_1 \| P_3 \approx P_2 \| P_3 \\ P_1 \approx P_2 & \textit{implies} & P_3 \| P_1 \approx P_3 \| P_2 \\ P_1 \approx P_2 & \textit{implies} & \textbf{hide } a_1, ..., a_n \textbf{ in } P_1 \approx \textbf{hide } a_1, ..., a_n \textbf{ in } P_2 \end{array}$$

Proof. We now prove that weak bisimilarity is substitutive with parallel composition. Let P_1 and P_2 be IOIMC with identical action signatures, let P be their union and let $x \approx_P y$. This

means that there is a weak bisimulation R on P such that $(x, y) \in R$. We now define the relation R' as follows:

$$R' = \{(x||z, y||z) \mid (x, y) \in R \wedge z \in S_{P_3}\}$$

Because R is an equivalence relation on P it is trivial that R' is also an equivalence relation on $P||P_3$.

Let $(s||u, t||u) \in R'$, then (s, t) is in R. Now let $s||u \stackrel{a}{\Longrightarrow} s'||u'$ with $a \in A^{all}$. This means that there exist $s''||u''$ and $s'''||u'''$ such that $s \Longrightarrow s''$, $u \Longrightarrow u''$, $s''||u'' \stackrel{a}{\dashrightarrow} s'''||u'''$, $s''' \Longrightarrow s'$ and $u''' \Longrightarrow u'$. From the definition of parallel composition we can deduce that $s''||u'' \stackrel{a}{\dashrightarrow} s'''||u'''$ implies that one of the following holds: $s'' \stackrel{a}{\dashrightarrow} s''' \wedge u''' = u'' \wedge a \in Act(P) \wedge a \notin Act(P_3)$ or $u'' \stackrel{a}{\dashrightarrow} u''' \wedge s''' = s'' \wedge a \in Act(P_3) \wedge a \notin Act(P)$ or $s'' \stackrel{a}{\dashrightarrow} s''' \wedge u'' \dashrightarrow u''' \wedge a \in Act(P) \wedge a \in Act(P_3)$. In other words either $s \stackrel{a}{\Longrightarrow} s' \wedge u \Longrightarrow u' \wedge a \in Act(P) \wedge a \notin Act(P_3)$ or $u \stackrel{a}{\Longrightarrow} u' \wedge s \Longrightarrow s' \wedge a \in Act(P_3) \wedge a \notin Act(P)$ or $s \stackrel{a}{\Longrightarrow} s' \wedge u \Longrightarrow u' \wedge a \in Act(P) \wedge a \in Act(P_3)$. Because $(s, t) \in R$ we find that there is a t' with $(s', t') \in R$ such that: $t \stackrel{a}{\Longrightarrow} t' \wedge u \Longrightarrow u' \wedge a \in Act(P) \wedge a \notin Act(P_3)$ or $u \stackrel{a}{\Longrightarrow} u' \wedge t \Longrightarrow t' \wedge a \in Act(P_3) \wedge a \notin Act(P)$ or $t \stackrel{a}{\Longrightarrow} t' \wedge u \Longrightarrow u' \wedge a \in Act(P) \wedge a \in Act(P_3)$. From the definition of parallel composition we now know that: $t||u \stackrel{a}{\Longrightarrow} t'||u'$ and from the definition of R' we know that $(s'||u', t'||u')$ is in R'. This means that the first clause of Definition 3.11 holds for R'.

From the definition of R' we can deduce that it has the following equivalence classes:

$$S^{all}/R' = \{\{x||y \mid x \in C\} \mid C \in S_P/R, y \in S_{P_3}\}$$

Because $(s, s') \in R$ if and only if $\forall u \in S_{P_3} \cdot (s||u, s'||u) \in R'$ we find that for an equivalence class $[s||u]_{R'}$: $[s||u]_{R'} = \{s'||u \mid s' \in [s]_R\}$.

Now let $(s||u, t||u) \in R'$ with $s||u \Longrightarrow s'||u'$ and $s'||u'$ stable and let C be an equivalence class of R' with $C \neq [s'||u']_{R'}$. From the definition of parallel composition we know that this means that $s \Longrightarrow s'$ and $u \Longrightarrow u'$ and both s' and u' stable. We know that each equivalence class of R' is derived from an equivalence class of R and a state in S_{P_3}. Let D be that equivalence class of R and let y be that state. So $C = \{s||y \mid s \in D\}$ and $u' = y \rightarrow D \neq [s']_R$. Note that if $u' \neq y$, $s'||u'$ cannot be in C. Now we find the following for the internal backward closure of C:

$$\begin{aligned} C^{int} &= \{x'||y' \mid \exists x||y \in C \cdot x'||y' \Longrightarrow x||y\} \\ &= \{x'||y' \mid \exists x \in D \cdot x' \Longrightarrow x \wedge y' \Longrightarrow y\} \end{aligned}$$

If we now look at the cumulative rate of $s'||u'$ to C^{int} we find:

$$\begin{aligned}
&\gamma_M(s'||u', C^{int}) \\
&= \sum\{|\lambda \mid \exists x \in D \cdot s' \xrightarrow{\lambda}{}^{M} x' \wedge x' \Longrightarrow x \wedge u' \Longrightarrow y|\} + \\
&\quad \sum\{|\lambda \mid \exists x \in D \cdot u' \xrightarrow{\lambda}{}^{M} y' \wedge s' \Longrightarrow x \wedge y' \Longrightarrow y|\} \\
&= \sum\{|\lambda \mid \exists x \in D \cdot s' \xrightarrow{\lambda}{}^{M} x' \wedge x' \Longrightarrow x \wedge u' = y|\} + \\
&\quad \sum\{|\lambda \mid \exists x \in D \cdot u' \xrightarrow{\lambda}{}^{M} y' \wedge s' = x \wedge y' \Longrightarrow y|\} \\
&\quad \text{(Because } s' \text{ and } u' \text{ are stable)} \\
&= \begin{cases} \gamma_M(s', D^{int}) + \gamma_M(u', \{y\}^{int}) & \text{, if } u' = y \wedge s' \in D \\ \gamma_M(s', D^{int}) & \text{, if } u' = y \wedge s' \notin D \\ \gamma_M(u', \{y\}^{int}) & \text{, if } u' \neq y \wedge s' \in D \\ 0 & \text{, if } u' \neq y \wedge s' \notin D \end{cases} \\
&= \begin{cases} \gamma_M(s', D^{int}) & \text{, if } u' = y \wedge s' \notin D \\ \gamma_M(u', \{y\}^{int}) & \text{, if } u' \neq y \wedge s' \in D \\ 0 & \text{, if } u' \neq y \wedge s' \notin D \end{cases} \\
&\quad \text{(Because } u' = y \rightarrow D \neq [s']_R \text{ means that } u' = y \wedge s' \in D \text{ is never true.)}
\end{aligned}$$

Because $(s,t) \in R$ we know that $s \Longrightarrow s'$ and s' stable imply that there is a t' such that $t \Longrightarrow t'$ and t' stable with $(s',t') \in R$[1] and $\gamma_M(s', D^{int}) = \gamma_M(t', D^{int})$ for every equivalence class D of R except $[s']_R$. It is furthermore trivial that s' and t' are in the same equivalence class of R. We now conclude that:

$$\begin{aligned}
&\gamma_M(s'||u', C^{int}) \\
&= \begin{cases} \gamma_M(s', D^{int}) & \text{, if } u' = y \wedge s' \notin D \\ \gamma_M(u', \{y\}^{int}) & \text{, if } u' \neq y \wedge s' \in D \\ 0 & \text{, if } u' \neq y \wedge s' \notin D \end{cases} \\
&= \begin{cases} \gamma_M(t', D^{int}) & \text{, if } u' = y \wedge t' \notin D \\ \gamma_M(u', \{y\}^{int}) & \text{, if } u' \neq y \wedge t' \in D \\ 0 & \text{, if } u' \neq y \wedge t' \notin D \end{cases} \\
&\quad (\gamma_M(t', D^{int}) = \gamma_M(s', D^{int}) \text{ because } s' \notin D \text{ implies that } D \neq [s']_R.) \\
&= \gamma_M(t'||u', C^{int})
\end{aligned}$$

So clause two of Definition 3.11 holds for R'. For any state $z \in S_{P_3}$ we find that if $(x,y) \in R$ then $(x||z, y||z) \in R'$ and since R' is a weak bisimulation $x||z \approx y||z$ thus proving that weak bisimilarity is substitutive with parallel composition.

[1] Because $(s,t) \in R$ and $s \Longrightarrow s'$ there is a y such that $t \Longrightarrow y$ and $(s',y) \in R$. For this pair clause 1 and 2 of weak bisimulation must hold once more. Eventually we will find t' this way.

We now prove that weak bisimilarity is substitutive with hiding. Let P be an IOIMC and let $s \approx_P t$ then there is a weak bisimulation R on P such that $(s,t) \in R$. Let $B \subseteq Ext(P)$ be a subset of the visual actions of P and let R' be a binary relation defined as follows:

$$R' = \{(hide\ B\ in\ s, hide\ B\ in\ t) \mid (s,t) \in R\}$$

It is trivial that R' is an equivalence relation on $hide\ B\ in\ P$. Let $hide\ B\ in\ s \overset{a}{\Longrightarrow} hide\ B\ in\ s'$. From the definition of hiding we can immediately deduce that there exists an $n \in \mathbb{N}$ with $n \geq 0$, a series of states $x_1, ..., x_n$ and a series of actions $b_1, ..., b_n \in B$ such that:

$$s \overset{b_1}{\Longrightarrow} x_1 \overset{b_2}{\Longrightarrow} ... x_k \overset{a}{\Longrightarrow} x_{k+1} ... \overset{b_{n-1}}{\Longrightarrow} x_n \overset{b_n}{\Longrightarrow} s'$$

Because $(s,t) \in R$ we find a series of states $x'_1, ..., x'_{n-1}$ such that:

$$t \overset{b_1}{\Longrightarrow} x'_1 \overset{b_2}{\Longrightarrow} ... x'_k \overset{a}{\Longrightarrow} x'_{k+1} ... \overset{b_{n-1}}{\Longrightarrow} x'_n \overset{b_n}{\Longrightarrow} t'$$

It follows that $hide\ B\ in\ t \overset{a}{\Longrightarrow} hide\ B\ in\ t'$ so the first clause of Definition 3.11 holds for R'.

Let $hide\ B\ in\ s \Longrightarrow hide\ B\ in\ s'$ and $hide\ B\ in\ s'$ stable. This means that there is an $n \in \mathbb{N}$ with $n \geq 0$, a series of actions $b_1, ..., b_n$ and a series of states $x_1, ..., x_n$ such that $s \overset{b_1}{\Longrightarrow} x_1 ... x_n \Longrightarrow s'$ with $b_1, ..., b_n \in B$. For t we then find the same series of actions and a weakly bisimilar set of states. We particularly find an x'_n such that $(x_n, x'_n) \in R$. Because $x_n \Longrightarrow s'$ and s' stable we find a t' such that $x'_n \Longrightarrow t'$ and t' stable with $\gamma_M(s', D) = \gamma_M(t', D)$ for all equivalence classes D of R except for $[s']_R$ and $(s', t') \in R$. Because $hide\ B\ in\ s'$ is stable s' and t' have no outgoing transitions labelled with an action in B.

From the definition of R' we see that its equivalence classes can be defined as follows:

$$\{\{hide\ B\ in\ x \mid x \in D\} \mid D \in S^{all}/R\}$$

Note that for any state x in S we find that $[hide\ B\ in\ x]_{R'} = \{hide\ B\ in\ y \mid y \in [y]_R\}$.

Let C be an equivalence class of R' such that $C \neq [hide\ B\ in\ s']_{R'}$ and let D be the corresponding equivalence class of R, thus $D \neq [s']_R$. We now find the following for the cumulative rate of $hide\ B\ in\ s'$ to C^{int}:

$$\begin{aligned}
&\gamma_M(hide\ B\ in\ s', C^{int}) \\
&= \{|\lambda \mid \exists (hide\ B\ in\ x') \in S^{all}, (hide\ B\ in\ x) \in C \cdot \\
&\quad hide\ B\ in\ s' \overset{\lambda}{-\!\!\!\!\rightarrow\!\!\!\!+}_M hide\ B\ in\ x' \wedge hide\ B\ in\ x' \Longrightarrow hide\ B\ in\ x|\} \\
&= \{|\lambda \mid \exists (hide\ B\ in\ x') \in S^{all}, x \in D \cdot \\
&\quad hide\ B\ in\ s' \overset{\lambda}{-\!\!\!\!\rightarrow\!\!\!\!+}_M hide\ B\ in\ x' \wedge hide\ B\ in\ x' \Longrightarrow hide\ B\ in\ x|\} \\
&= \{|\lambda \mid \exists x' \in S^{all}, x \in D \cdot s' \overset{\lambda}{-\!\!\!\!\rightarrow\!\!\!\!+}_M x' \wedge hide\ B\ in\ x' \Longrightarrow hide\ B\ in\ x|\} \\
&= \gamma_M(s', D^{int+})
\end{aligned}$$

Here $D^{int+} = \{x' \mid \exists x \in D \cdot hide\ B\ in\ x' \Longrightarrow hide\ B\ in\ x\}$. Note that $D^{int+} \supseteq D^{int}$. For some $n \in \mathbb{N}$ with $n \geq 0$ let $E_1, ..., E_n$ be the equivalence classes of R that can reach D with a number of weak

transitions labelled with actions in B or with a weak move (note that this means that D itself is always in $E_1, ..., E_n$). In other words we find that:

$$\forall E_x \cdot (\forall e \in E_x \cdot \exists m \in \mathbb{N}, f_1, ..., f_m \in S^{all}, b_1, ..., b_m \in B \cdot \\ m \geq 0 \ \wedge \ e \overset{b_1}{\Longrightarrow} f_1 ... f_{m-1} \overset{b_m}{\Longrightarrow} f_m \ \wedge \ f_m \in D)$$

When we hide the set of actions B in the states of the equivalence classes E_x we find that they move internally to states in equivalence class C. Note that because *hide B in s′* is stable and $C \neq [\textit{hide } B \textit{ in } s']_{R'}$, $[s']_R$ can not be in E (that would either mean that *hide B in s′* moves internally to a state in C making it unstable or that *hide B in s′* is in C but this is not possible because $C \neq [\textit{hide } B \textit{ in } s']_{R'}$). So $D^{int+} = \bigcup_{E_i} E_i^{int}$. Now we find that $\gamma_M(s', D^{int+}) = \gamma_M(t', D^{int+})$ (Recall that we found earlier that $\gamma_M(s', D) = \gamma_M(t', D)$ for all equivalence classes D of R except $[s']_R$ which is not in E). In the same way we showed that $\gamma_M(\textit{hide } B \textit{ in } s', C)$ equals $\gamma_M(s', D^{int+})$, we can now show that $\gamma_M(t', D^{int+})$ equals $\gamma_M(\textit{hide } B \textit{ in } t', C^{int})$. This proves that the second clause of Definition 3.11 holds for R'. This means that *hide B in s* $\approx$ *hide B in t* when $s \approx t$ and thus weak bisimilarity is substitutive with hiding.

Appendix B

Complete IOIMC models of DFT elements

In this appendix the complete IOIMC models of DFT elements are given. For each DFT element we will give the IOIMC action signature and its behavior in both IML and schematic form based on its actions and delay-rates. The *IML* language is given in Section B.1. Together the action signature and behavior define the IOIMC as in Definition 3.2. The action signatures are based on the immediate surroundings of the corresponding DFT element. This translation can be found in Section 4.7. In the schematic representation of the IOIMC behavior dotted lines between two transitions or states represent a finite number of similar transitions or states. Both the IML and schematic representations of the behaviors omit input-actions from a state to itself. All models are input-enabled and the true behaviors can be found by adding a transition to itself for any state that is missing an outgoing input action. This transition should of course be labelled with the missing input action.
The schematic representations of the IOIMC models given in this appendix are often recursive. In these schematic representations states have names and parameters in the form of actions. Dotted states represent states that are already present in the diagram only with different parameters. For instance, in Figure B.2, the transition from state $AND(f(M), a(M), F_i)$ to dotted state $AND(f(M), a(M), F_i \setminus f_1)$ labelled $f(i_1)$? should be read as a transition from the starting state to itself, except with one element of the vector F_i removed. Intuitively this means that, after one of its inputs fires, an n-input AND-gate behaves the same as an (n-1)-input AND-gate. Dotted lines between two states, for instance state $AND(f(M), a(M), F_i \setminus f_1)$ and $AND(f(M), a(M), F_i \setminus f_n)$ in Figure B.2, denote a finite number of states. The same goes for dotted lines between transitions.

B.1 Notation

To make talking about IOIMC somewhat easier we now define the language IML [15, section 5.1]. We will use this language to describe IOIMC textually. We assume a countable set of variables $\mathcal{V}$ that will be used to express repetitive behavior.

Definition B.1 *Let $\lambda \in \mathbb{R}^+$, $a \in Act$ and $X \in \mathcal{V}$. We define the language* IML *as the set of expressions given by the following grammar.*

$$\mathcal{E} ::= 0 \mid a.\mathcal{E} \mid (\lambda).\mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid X \mid \underline{x:=\mathcal{E}} \mid \bot$$

The intuitive meaning of the language constricts is as follows:

- 0 describes a terminated behavior that cannot perform any interactive- or Markovian transition.
- $a.E$ describes an IOIMC that, after performing interaction a, will behave like the IOIMC described by E. We say that E is *action prefixed* by a.
- $(\lambda).E$ describes an IOIMC that will behave like IOIMC E after a delay which is exponentially distributed with rate λ. We say that E is *delay prefixed* by λ.
- $E + F$ describes two alternatives. The IOIMC $E + F$ may behave like IOIMC E or like IOIMC F. How this choice is made depends on the IOIMC E and F.
- $\underline{x:=E}$ describes a recursive behavior. $\underline{x:=E}$ behaves like IOIMC E, but when the variable X is encountered it reinitializes to $\underline{x:=E}$.

B.2 Cold Basic Event

Action signature

For IOIMC $CBE(f_M, a_M, rate)$ we find action signature Sig with:
– $in(Sig) = \{a_M\}$
– $out(Sig) = \{f_M\}$
– $int(Sig) = \emptyset$

IML definition

$$\begin{aligned} CBE(f_M, a_M, rate) &= a_M?.CBE_{active}(f_M, rate) \\ CBE_{active}(f_M, rate) &= (rate).CBE_{firing}(f_M) \\ CBE_{firing}(f_M) &= f_M!.0 \end{aligned}$$

Schematic of behavior

A schematic of the behavior of a cold basic event can be found in figure 4.7.

B.3 Warm Basic Event

Action signature

For IOIMC $WBE(f_M, a_M, rate_A, rate_D)$ we find action signature Sig with:
– $in(Sig) = \{a_M\}$
– $out(Sig) = \{f_M\}$

– $int(Sig) = \emptyset$

IML definition

$$\begin{aligned} WBE(f_M, a_M, rate_A, rate_D) &= a_M?.WBE_{active}(f_M, rate_A) + (rate_D).WBE_{firing}(f_M) \\ WBE_{active}(f_M, rate_A) &= (rate_A).WBE_{firing}(f_M) \\ WBE_{firing}(f_M) &= f_M!.0 \end{aligned}$$

Schematic of behavior

A schematic of the behavior of a cold basic event can be found in Figure 4.8.

B.4 Hot Basic Event

Action signature

For IOIMC $HBE(f_M, a_M, rate)$ we find action signature Sig with:
– $in(Sig) = \{a_M\}$
– $out(Sig) = \{f_M\}$
– $int(Sig) = \emptyset$

IML definition

$$\begin{aligned} HBE(f_M, a_M, rate) &= a_M?.HBE_{active}(f_M, rate) + (rate).HBE_{firing}(f_M) \\ HBE_{active}(f_M, rate) &= (rate).HBE_{firing}(f_M) \\ HBE_{firing}(f_M) &= f_M!.0 \end{aligned}$$

Schematic of behavior

A schematic of the behavior of a cold basic event can be found in Figure 4.9.

B.5 OR-gate

Action signature

For IOIMC $OR(f_M, a_M, F_i)$ we find action signature Sig with:
– $in(Sig) = \{f_i \mid f_i \in F_i\}$
– $out(Sig) = \{f_M\}$

– $int(Sig) = \emptyset$

IML definition

$$OR(f_M, a_M, F_i) \quad = \quad \sum_{f_i}^{f_i \in F_i} f_i?.f_M!.0$$

Schematic of behavior

A schematic representation of the behavior of an OR-gate can be found in Figure B.1.

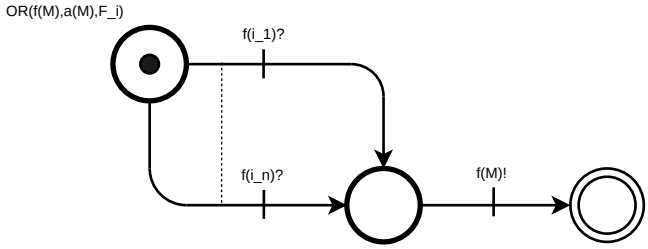

Figure B.1: Schematic of the behavior of an OR-gate.

B.6 AND-gate

Action signature

For IOIMC $AND(f_M, a_M, F_i)$ we find action signature Sig with:
– $in(Sig) = \{f_i \mid f_i \in F_i\}$
– $out(Sig) = \{f_M\}$
– $int(Sig) = \emptyset$

IML definition

$$AND(f_M, a_M, F_i) \quad = \quad \begin{cases} f_M!.0 & \text{, if } F_i = \langle\rangle \\ \sum_{f_i}^{f_i \in F_i} f_i?.AND(f_M, a_M, F_i \setminus \langle f_i \rangle) & \text{, otherwise} \end{cases}$$

Schematic of behavior

A schematic representation of the behavior of an AND-gate can be found in Figure B.2.

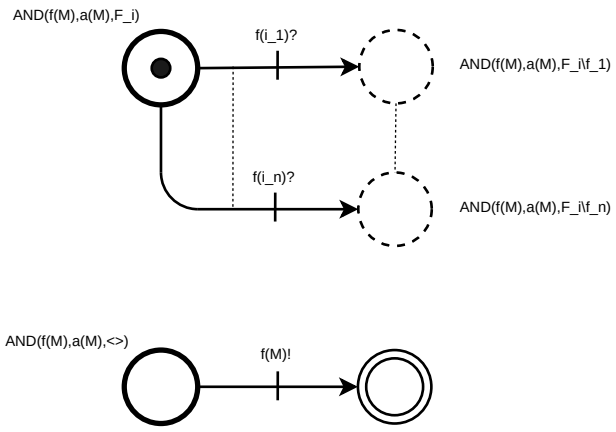

Figure B.2: Schematic of the behavior of an AND-gate.

B.7 K/M-gate

Action signature

For IOIMC $KM(f_M, a_M, k, m, F_i)$ we find action signature *Sig* with:
– $in(Sig) = \{f_i \mid f_i \in F_i\}$
– $out(Sig) = \{f_M\}$
– $int(Sig) = \emptyset$

IML definition

$$KM(f_M, a_M, k, m, F_i) = \begin{cases} f_M!.0 & \text{, if } |F_i| \leq m - k \\ \sum_{f_i}^{f_i \in F_i} f_i?.KM(f_M, a_M, k, m, F_i \setminus \langle f_i \rangle) & \text{, otherwise} \end{cases}$$

Schematic of behavior

A schematic representation of the behavior of a K/M-gate can be found in Figure B.3.

B.8 PAND-gate

Action signature

For IOIMC $PAND(f_M, a_M, F_i)$ we find action signature *Sig* with:
– $in(Sig) = \{f_i \mid f_i \in F_i\}$
– $out(Sig) = \{f_M\}$

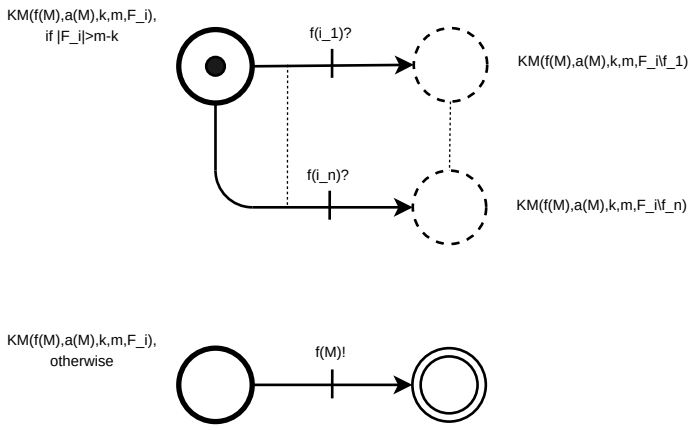

Figure B.3: Schematic of the behavior of a K/M-gate.

– $int(Sig) = \emptyset$

IML definition

$$PAND(f_M, a_M, F_i) =$$
$$\begin{cases} f(A)!.0 & \text{, if } F_i = \langle\rangle \\ \textbf{head } F_i?.PAND(f_M, a_M, \textbf{tail } F_i) + \sum_{f_i}^{f_i \in \textbf{tail } F_i} f_i?.0 & \text{, otherwise} \end{cases}$$

Schematic of behavior

A schematic representation of the behavior of a PAND-gate can be found in Figure B.4.

B.9 Spare gate

Action signature

For IOIMC $SG(f_M, a_M, f_{pr}, \vec{S})$ we find action signature Sig with:
– $in(Sig) = \{a_M\} \cup \left(\bigcup_{U_s}^{(a_s,U_s)\in\vec{S}} \bigcup_{u_s}^{u_s\in U_s} u_s \right)$
– $out(Sig) = \{f_M\} \cup \left(\bigcup_{a_s}^{(a_s,U_s)\in\vec{S}} a_s \right)$
– $int(Sig) = \emptyset$

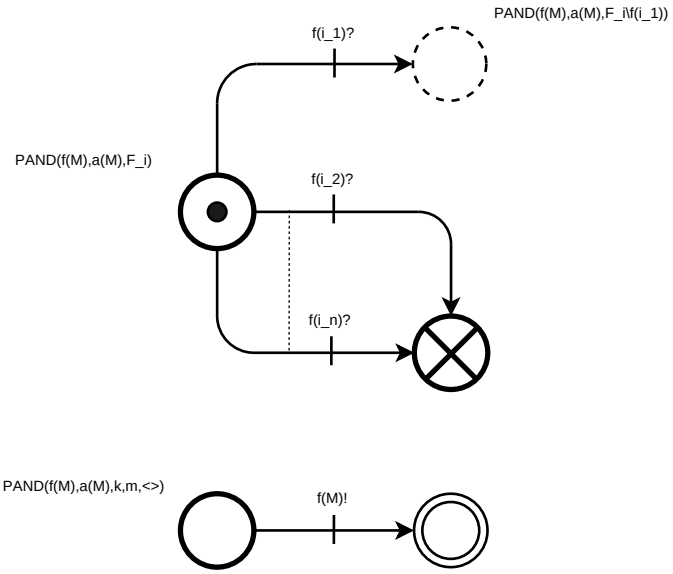

Figure B.4: Schematic of the behavior of an PAND-gate.

IML definition

$$
\begin{aligned}
&SG(f_M, a_M, f_{pr}, \vec{S}) = \\
&\quad SG_{D,P}(f_M, a_M, f_{pr}, \vec{S}) \\
&SG_{D,P}(f_M, a_M, f_{pr}, \vec{S}) = \\
&\quad f_{pr}?.SG_{D,N}(f_M, a_M, \vec{S}) + \\
&\quad \sum_{U_s}^{(a_s,U_s)\in\vec{S}} \left(\sum_{u_s}^{u_s\in U_s} u_s?.SG_{D,P}(f_M, a_M, f_{pr}, \vec{S} \setminus \langle (a_s, U_s) \rangle) \right) + \\
&\quad a_M?.SG_{A,P}(f_M, f_{pr}, \vec{S}) \\
&SG_{D,N}(A, ST, \vec{S}) = \\
&\quad \begin{cases} \sum_{U_s}^{(a_s,U_s)\in\vec{S}} \left(\sum_{u_s}^{u_s\in U_s} u_s?.SG_{D,N}(f_M, a_M, \vec{S} \setminus \langle (a_s, U_s) \rangle) \right) + & \\ a_M?.SG_{A,N}(f_M, \vec{S}) & \text{, if } \vec{S} = \langle\rangle \\ f_M!.0 & \text{, otherwise} \end{cases} \\
&SG_{A,P}(f_M, f_{pr}\vec{S}) = \\
&\quad f_{pr}?.SG_{A,N}(f_M, \vec{S}) + \\
&\quad \sum_{U_s}^{(a_s,U_s)\in\vec{S}} \left(\sum_{u_s}^{u_s\in U_s} u_s?.SG_{A,P}(f_M, f_{pr}, \vec{S} \setminus \langle (a_s, U_s) \rangle) \right) \\
&SG_{A,N}(A, \vec{S}) = \\
&\quad \begin{cases} \textbf{first head } \vec{S}!.SG_{A,P}(f_M, \textbf{head second head } \vec{S}, \textbf{tail } \vec{S}) + & \\ \sum_{U_s}^{(a_s,U_s)\in\vec{S}} \left(\sum_{u_s}^{u_s\in U_s} u_s?.SG_{A,N}(f_M, \vec{S} \setminus \langle (a_s, U_s) \rangle) \right) & \text{, if } |\vec{S}| > 0 \\ f_M!.0 & \text{, otherwise} \end{cases}
\end{aligned}
$$

Remember that $\vec{S}$ consists of tuples of an activation signal and a vector of disabling signals for each of the spare gate's spares. So, **first head** $\vec{S}$! denotes the activation signal of the first spare and **head second head** $\vec{S}$ denotes the firing signal of this spare since the vector of disabling signals for each spare always starts with the spare's firing signal.

Schematic of behavior

A schematic representation of the behavior of a spare gate can be found in Figure B.5.

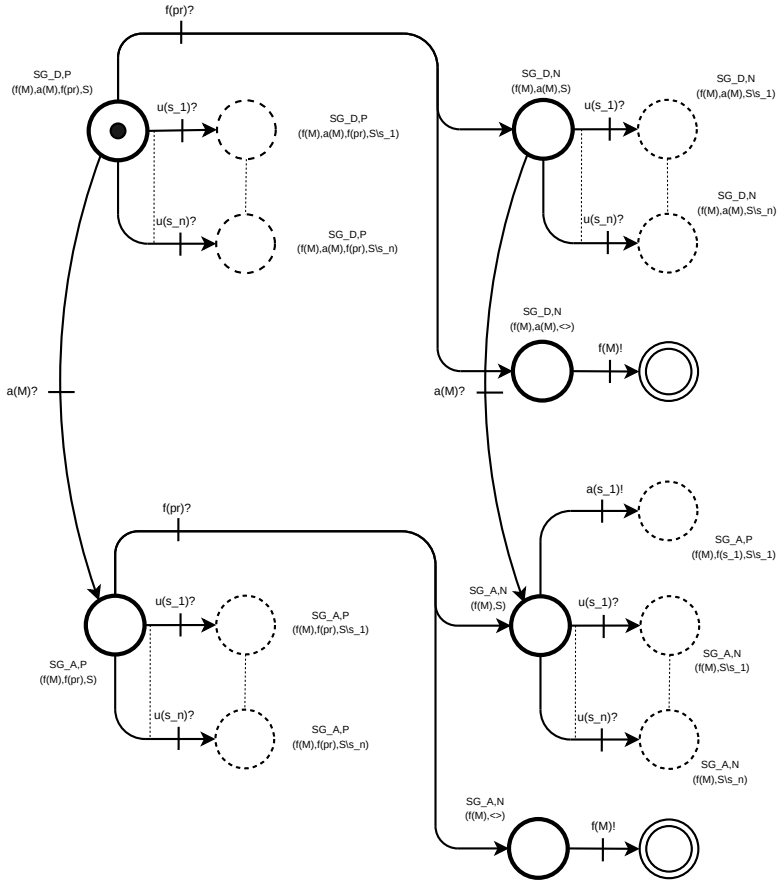

Figure B.5: Schematic of the behavior of a spare gate.

B.10 Activation Auxiliary

Action signature

For IOIMC $AA(a_M, A_M)$ we find action signature Sig with:
– $in(Sig) = \{a_{x,M} \mid a_{x,M} \in A_M\}$
– $out(Sig) = \{a_M\}$

– $int(Sig) = \emptyset$

IML definition

$$AA(a_M, A_M) \quad = \quad \sum_{a_{x,M}}^{a_{x,M} \in A_M} a_{x,M}?.a_M!.0$$

Schematic of behavior

A schematic representation of the behavior of an activation auxiliary can be found in Figure B.6.

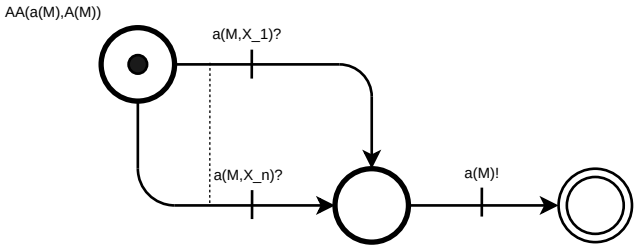

Figure B.6: Schematic of the behavior of an activation auxiliary.

B.11 Firing Auxiliary

Action signature

For IOIMC $FA(f_M, F'_M)$ we find action signature Sig with:
– $in(Sig) = \{f'_M \mid f'_M \in F'_M\}$
– $out(Sig) = \{f_M\}$
– $int(Sig) = \emptyset$

IML definition

$$FA(f_M, F'_M) \quad = \quad \sum_{f'_M}^{f'_M \in F'_M} f'_M?.f_M!.0$$

Schematic of behavior

A schematic representation of the behavior of a firing auxiliary can be found in Figure B.7.

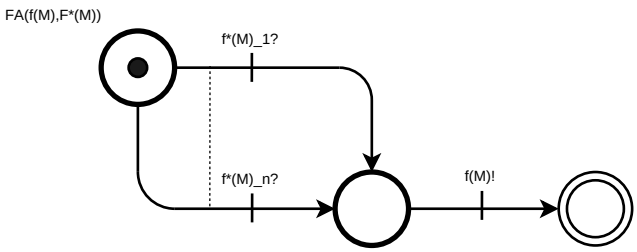

Figure B.7: Schematic of the behavior of an activation auxiliary.

Bibliography

[1] A Aziz, K Sanwal, V Singhal, and R K Brayton. Verifying continuous-time markov chains. In *Eighth International Conference on Computer Aided Verification*, volume 1102, pages 269–276. Springer Verlag, 1996.

[2] Galileo DFT analysis tool. http://www.cs.virginia.edu/˜ftree. Galileo Website.

[3] C Baier, B Haverkort, H Hermanns, and J-P Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[4] C Baier, H Hermanns, J-P Katoen, and V Wolf. Comparative branching-time semantics. In *CONCUR*, pages 492–507, 2003.

[5] A Bobbio, G Franceschinis, R Gaeta, and L Portinale. Parametric fault tree for the dependability analysis of redundant systems and its high-level petri net semantics. *IEEE Transactions on Software Engineering*, 29(33):270–287, 2003.

[6] H Boudali. *A Temporal Bayesian Network Reliability Modeling and Analysis Framework*. PhD thesis, University of Virginia, 2005.

[7] M A Boyd and S J Bavuso. Simulation modeling for long duration spacecraft control systems. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 106–113, 1993.

[8] Mario Bravetti. Revisiting interactive markov chains. *Electr. Notes Theor. Comput. Sci.*, 68(5), 2002.

[9] D Coppit and K J Sullivan. Formal specification in collaborative design of software tools. *Submitted to High Assurance System Engineering (HASE) Conference*, 1999.

[10] O Coudert and J C Madre. Fault tree analysis: 10^{20} prime implicants and beyond. In *Proceedings of the Annual Reliability and Maintainability Symposium*, 1993.

[11] J B Dugan, S J Bavuso, and M A Boyd. Dynamic fault tree models for fault tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–377, September 1992.

[12] J B Dugan and K S Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Transactions on Computers*, 38(6):775–787, 1989.

[13] Y Dutuit and A Rauzy. A linear time algorithm to find modules of fault trees. *IEEE Transactions on Reliability*, 45(3):422–425, 1996.

[14] R Gulati and J B Dugan. A modular approach for analyzing static and dynamic fault trees. In *Proceedings of the Annual Reliability and Maintanability Symposium*, pages 69–75, 1998.

[15] H Hermanns. *Interactive Markov Chains*, volume 2428 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[16] H Hermanns, U Herzog, U Klehmet, V Mertsiotakis, and M Siegle. Compositional performance modelling with the tipptool. *Performance Evaluation*, 39(1-4):5–35, 2000.

[17] H Hermanns and J-P Katoen. Automated compositional Markov chain generation for a plain-old telephone system. *Science of Computer Programming*, 36(1):97–127, 2000.

[18] R A Howard. *Dynamic probability systems. Volume 1: Markov models*. Decision and Control. John Wiley & Sons, Inc., 1971.

[19] P A Lee and T Anderson. *Fault tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant systems*. Prentice Hall, 1981.

[20] J Levine, T Mason, and D Brown. *Lex & Yacc*. O'Reilly, second edition edition, 1992.

[21] N A Lynch and M R Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1988.

[22] M. Malhotra and K. S. Trivedi. Dependability modeling using petri-nets. *IEEE Transactions on Reliability*, 44(3):428–440, September 1995.

[23] R Manian, J B Dugan, D Coppit, and K J Sullivan. Combining various solution techniques for dynamic fault tree analysis of computer systems. *IEEE International High-Assurance Systems Engineering Symposium*, 3:21–28, 1998.

[24] R Milner. *Communication and Concurrency*. Prentice Hall Inc., 1989.

[25] S. Montani, L. Portinale, A. Bobbio, and D. C. Raiteri. Automatically translating dynamic fault trees into dynamic bayesian networks by means of a software tool. In *Proceedings of The First International Conference on Availability, Reliability and Security (ARES)*, pages 804–809. IEEE Computer Society, 2006.

[26] D C Raiteri, M Iacono, G Franceschinis, and V Vittorini. Repairable fault tree for the automatic evaluation of repair policies. *International Conference on Dependable Systems and Networks*, pages 659–668, 2004.

[27] A Rauzy. New algorithms for fault tree analysis. *Reliability Engineering and System Safety*, 40:203–211, 1993.

[28] W J Stewart. *Matrix-Geometric Solutions in Stochastic Models, An Algorithmic Approach*. The John Hopkins University Press, 1981.

[29] W J Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[30] K K Vemuri, J B Dugan, and K J Sullivan. Automatic synthesis of fault trees for computer-based systems. *IEEE Transactions on Reliability*, 48(4):394–402, 1999.

[31] W E Veseley, F F Goldberg, N H Roberts, and D F Haasl. *Fault Tree Handbook*, volume (NUREG-0492). United States Nuclear Regulatory Commision, 1981.

[32] H A Watson and Bell Telephone Laboratories. *Launch Control Safety Study*. Bell Telephone Laboratories, 1961.

[33] D J White. *Markov Decision Processes*. John Wiley & Sons Ltd., 1993.