A New Component Concept for Fault Trees

Bernhard Kaiser[1], Peter Liggesmeyer[1], Oliver Mäckel[2]

[1]Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam,
Dept. for Software Engineering and Quality Management, Prof.-Dr.-Helmert-Str. 2-3, Potsdam, Germany
[2]Siemens AG, Corporate Technology, Simulation & Risk Management, Otto-Hahn-Ring 6, Munich, Germany

{bernhard.kaiser|peter.liggesmeyer}@hpi.uni-potsdam.de, oliver.maeckel@siemens.com

Abstract

The decomposition of complex systems into manageable parts is an essential principle when dealing with complex technical systems. However, many safety and reliability modelling techniques do not support hierarchical decomposition in the desired way. Fault Tree Analysis (FTA) offers decomposition into modules, a breakdown with regard to the hierarchy of failure influences rather than to the system architecture. In this paper we propose a compositional extension of the FTA technique. Each technical component is represented by an extended Fault Tree. Besides the internal basic events and gates, each component can have input and output ports. By connecting these ports, components can be integrated into a higher-level system model. All components can be developed independently and stored in separate files or component libraries. Mathematically, each Component Fault Tree represents a logical function from its input ports and internal events to its output ports. As in traditional FTA, both qualitative and quantitative analyses are possible. Known algorithms e.g. based on Binary Decision Diagrams (BDDs) can still be applied. The Windows based safety analysis tool UWG3 has been developed to prove this concept in practice. It allows creating component libraries in an exchangeable XML format. We have carried out some case studies in order to show that the new concept improves clearness and intuitive modelling while maintaining the same results as traditional FTA.

Keywords: Fault Trees, Safety, Reliability

1 Introduction

Today's technical systems typically consist of hardware and software and have grown too complex that a single person is able to understand them as a whole. Hierarchical decomposition is the underlying principle in understanding complex systems. In hierarchical models, a system consists of components, which are recursively refined into sub-components. A set of rules allows determining the properties of the whole system based on the properties of its components and its architecture. Models that allow this proceeding are called compositional models. Many design models used in industry or in academic research provide compositional semantics (Harel 1987, Clarke and Wing 1996).

For many technical systems, safety and reliability are important quality aspects. Therefore, there is a strong demand for techniques and models that help the developer to achieve and to assess these properties. Some of these techniques are Fault Tree Analysis (FTA) (IEC 61025, DIN 25424, Vesely 1981), Event Tree Analysis, or Reliability Block Diagrams (RBDs). Some of them also allow the examination of a technical system from the component perspective, similarly to design models. The components, however, are not necessarily the same that have been identified during the design process. We will examine this in the case of Fault Trees.

In FTA causal chains leading to some failure are depicted as a tree. The system failure to be examined is the root of the tree; the basic influence factors are the leaves. This tree structure inherently describes a hierarchical breakdown, but with regard to the hierarchy of failure influences rather than to the system architecture. A notion of modules does exist in FTA; it is used with the meaning of independent subtrees. This partition merely represents a property of the influence chains. The modules generally do not correspond to the technical components that have been identified during system development. Technical Components are often influenced by other components and thus are no modules. So there is no way to assign to each technical component a separate and reusable entity in FTA. As a consequence it is not possible to utilise several developers to construct partial FTA models for later integration into an overall system model. Moreover the integration of Fault Trees with the models obtained from the system design phase or the automatic generation of Fault Trees from design artefacts is not directly possible. Another issue is that the tree structure is sometimes insufficient to model failure propagation paths since common cause failures influence the top-event by more than one path. Therefore they must be split into several "repeated events" in order to preserve the tree structure.

To overcome these drawbacks we propose to extend the traditional Fault Trees with a notion of components that are connected via ports. These components need not be modules in the sense of independent subtrees. They can be partitioned according to the real components of the technical system. It is possible to elaborate and store them independently. It is only when the analysis is started that all the component models involved need to be available. In previous work (Mäckel, Liggesmeyer 2000), we extended the trees to Directed Acyclic Graphs (DAGs),

which allow the resolution of common failure dependencies and negate the need for repeated events.

In a co-operation between the Hasso-Plattner-Institute, Siemens and DaimlerChrysler, we implemented this extended FTA technique in the safety and reliability analysis tool UWG3. This tool has proven its intuitive effectiveness in several case studies and some first industrial applications.

In Section 2 of this paper we give a short overview over classical FTA, the traditional notion of hierarchy and present some of the resulting drawbacks. We also present our previous work on extending the tree topology to a Directed Acyclic Graph. In Section 3 we introduce our new component concept that extends classical Fault Trees. We intuitively develop it step by step and finally give a formal definition. In Section 4 we present our tool UWG3 and report about its practical application. Section 5 gives some concluding remarks and provides an outlook on our future projects.

2 State of the Art and Previous Work

2.1 Traditional Fault Tree Analysis

FTA is an analysis technique for safety and reliability aspects that uses a graphical representation to model causal chains leading to failures. It was first invented in 1961 for the Minuteman Launch Control System, then further developed and defined in international standards (DIN 25424, IEC 61025) and other literature as the NUREG Fault-Tree-Handbook (Vesely 1981). The concept is to start with a failure event or hazard state and to trace its influences back until the basic influence factors are reached. The resulting influence hierarchy is depicted as an upside-down tree with the failure event (referred to as "top-event") at its root. Mainly two connectives are used to express how influences contribute to a consequential failure:

- the AND connective, indicating that all influence factors must apply simultaneously, and
- the OR connective, indicating that at least one of the influence factors must apply to cause the failure

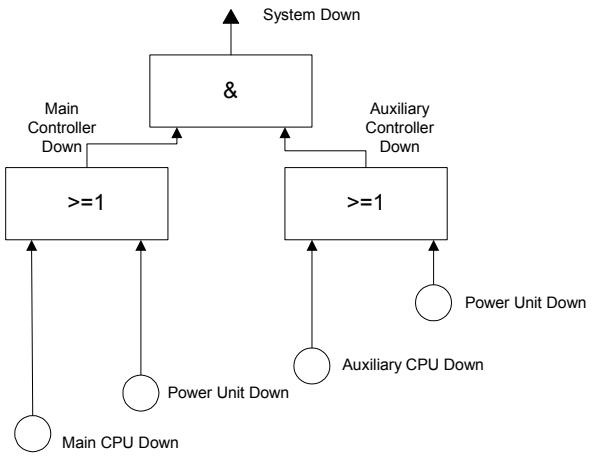

Fig. 2-1: Simple Fault Tree

Fig. 2-1 provides an example of the Fault Tree graphical notation. In this example, we imagine some Controller System that has a redundant structure consisting of a Main Controller and an Auxiliary Controller. The whole system is unavailable if both Controllers are unavailable at the same time. Further, each Controller is down if either its corresponding CPU is down or the Power Unit is down (or both).

The Fault Tree is the parse tree of the corresponding propositional logic formula:

SystemDown=(MainCPUDown $\vee$ PowerUnitDown) $\wedge$ (AuxCPUDown $\vee$ PowerUnitDown)

SystemDown is the top-event; the four events on the bottom line are called basic events. The boxes representing the propositional logic connectives are called gates (the term stems from integrated logic circuits).

The analyses to be performed on this Fault Tree can be either qualitative or quantitative. Qualitative analyses show, for instance, which combinations of failures must occur together to cause a top-level failure. Quantitative analysis, on the other hand, calculates the probability of the top event occurring from the probabilities of the basic events. It is important to know that most calculation rules for the probabilistic analysis depend on the assumption that all events are stochastically independent of each other. Many modern analysis algorithms make use of the efficient coding of Boolean formulae by Binary Decision Diagrams (BBDs) (Bryant 1986, Doyle and Dugan 1995, Coudert and Madre 1993) and of corresponding manipulation techniques.

Besides AND and OR Fault Trees allow the use of Exclusive-OR (sometimes denoted XOR), NOT, INHIBIT, and n-out-of-m-VOTER gates. However, we will confine ourselves to AND and OR for the following discussion. We point out that the presence of NOT gates has no impact on our concept.

2.2 Generation of Fault Trees

Fault trees are usually generated manually. Highly skilled, experienced engineers analyse the system. Considerable knowledge, system insight and overview are necessary to consider various failure modes and their consequences at a time. This manual work is error-prone, requires substantial effort, and is still often incomplete, since no single person can comprehend the whole system. These disadvantages can be avoided by the automation of Fault Tree generation. During the development of a system, documents are produced that contain information about the system's behaviour and structure (e.g. formal specifications, circuit designs and source code). These documents implicitly contain information about misbehaviours and can be used to partly automate the Fault Tree generation. Because of this we have focused on the automatic generation of Fault Trees for hardware and software units (Liggesmeyer and Rothfelder 1998, Liggesmeyer and Mäckel 2001), and tools for integration and analysis (Mäckel and Rothfelder2001).

Fault trees of complex systems are often very large – especially generated ones. Ideally, the modularity of

complex systems should be reflected by a corresponding structure of the Fault Tree. The known module concept for Fault Trees is often not appropriate to do so. This is due to the fact that, in practice, many causes are contained in several subtrees. For this reason, the precondition of statistical independence of the causes in standard Fault Trees is no longer true. Neglecting this circumstance leads to incorrect results during calculation of reliability characteristics. Currently, this problem is handled by marking multiple occurrences of causes as *repeated events*.

2.3 Cause Effect Graphs

In some cases two (or more) branches of the Fault Tree depend on the same basic failure event. For instance, looking at the example from Fig. 2-1 we may ask the question if each Controller has a Power Unit on its own or if there is a common Power Unit feeding both of them (see Fig. 2-2).

In the latter case the independence of basic events is no longer given. The calculation algorithm must know about the repeated events; otherwise it will produce incorrect results.

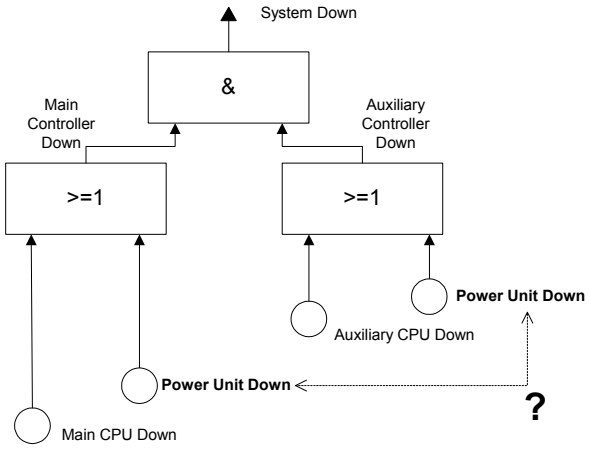

Fig. 2-2: Repeated Event

Practical experience (Mäckel and Rothfelder 2001) shows that Fault Trees usually contain a large number of *repeated events*. There are computation methods, based on minimal cut sets (Kececioglu 1991) or on BDDs, that can handle *repeated events* correctly. Depending on the FTA tool either the user marks events as repeated or the tool considers events that have the same name as repeated events. In either case there is some danger of confusion (since in large systems it is possible that events accidentally have the same name) or of inconsistencies. Sometimes the user even wishes to give the same name to equal failures in different technical components.

Apart from repeated events, the fact that Fault Trees contain only one top-event (IEC 61025) is also a restriction. In practice, it is often important to analyse cause-effect relations between various top-events that represent different failure modes of the same technical component.

To overcome both problems, we previously defined a generalized form of Fault Trees (Mäckel and Rothfelder 2001). We call these diagrams Cause Effect Graphs (CEGs). CEGs are Directed Acyclic Graphs (DAGs or Acyclic Digraphs). The following differences exist between Fault Trees and CEGs:

- *Repeated events* are represented only once (see Fig. 2-3 in comparison to Fig. 2-2). This reduces the size of the graph and improves readability and maintainability. It is much easier to keep the Fault Tree consistent during modifications. Furthermore, CEGs may also be read bottom-up. It is easy to determine all the effects a specific cause is correlated with. This is not directly possible in standard Fault Trees, because it would be necessary to identify all instances of the causes.
- CEGs may contain several top-events (see Fig. 2-4). This permits to take into account different undesired events during system optimisation. Further, it is possible to analyse relations between different top-events within a system.

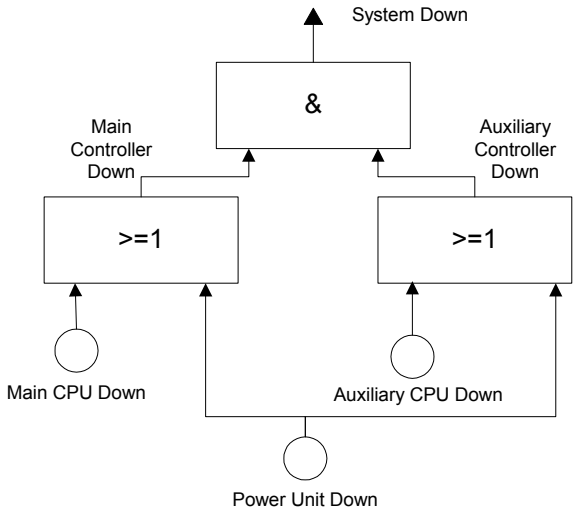

Fig. 2-3: Cause Effect Graph

Although the logical structure of the above example is no longer a tree but a DAG, it is still possible to represent the corresponding Boolean formula by a BBD and to calculate the top event probability by the standard algorithms.

To give an example for the simplification obtained, a CEG for an automotive system contained 200 instead of the 9000 elements that are contained in the corresponding standard Fault Tree representation. There were about 150 instances of just one specific *repeated event*.

By allowing more than one top-event it is possible to examine several failure modes and their common influences at a time. A cause that is connected to several top-events is usually an appropriate starting point for system optimisation. Some kinds of analyses that have traditionally been carried out by Event Tree Analysis can now be performed using CEGs.

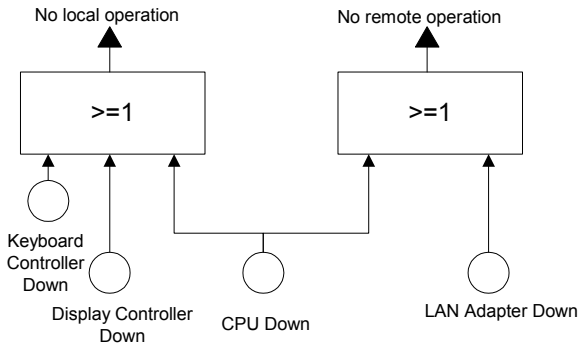

Fig. 2-4: Two Top-Events

The transition from trees to DAGs is a precondition and support for the new component concept presented in this paper. As we will see, the top events of CEGs will appear there as output ports of the corresponding Component Fault Trees. In the following we will use the terms Fault Tree and Cause Effect Graph interchangeably.

2.4 Hierarchy and Modules in Fault Trees

As stated before, complex Fault Trees or CEGs need to be partitioned both for editing and for efficient computer analysis. The goal is to generate separate Fault Trees for different system components, e.g. electronic components and software, and then to combine the Fault Trees in order to get a valid analysis result for the system.

Today, the usual principle for handling complexity in Fault Trees is division into independent subtrees, called modules. A module is a subtree that is not influenced by other parts of the Fault Tree and influences other parts of the tree only by its root (DIN25424, Kohda et al 1989).

Modularisation is a recursive process as subtrees might themselves contain independent subtrees. In particular the whole Fault Tree and every basic event are independent modules.

Traditionally, the top-event probability of each module is calculated and then the whole module is replaced by a virtual simple event, located where the subtree root was before. This virtual event has the subtree root probability assigned to it. This continues until the top-event probability of a Fault Tree has been calculated. So for probabilistic analysis the output of the root gate of a module is handled like a basic event.

Identification of modules in Fault Trees is a formal procedure that only refers to the tree structure and not to the system architecture. Due to *repeated events* and influences between technical components, it is often necessary for events belonging to different technical units to be members of the same module. Thus, system architecture components do not necessarily correspond directly to Fault Tree modules.

The refinement of a system into components by its architecture is a different kind of hierarchy and, as we have seen, not necessarily related to the refinement into modules. In summary, we find two distinct refinement hierarchies in Fault Trees:

1. the backward refinement of the cause-effect relations as indicated by the tree
2. the refinement by architectural components. (Mäckel and Rothfelder 2001).

In Section 3 we introduce a method for decomposing a Cause Effect Graph based on architectural components.

3 Fault Trees with Hierarchical Components

3.1 Informal Introduction

To introduce our component concept intuitively, we start by revisiting the CEG example from Fig. 2-3. The events shown can be related, for instance, to three different technical units:

1. The Main Controller
2. The Auxiliary Controller
3. The Power Unit

In the following we will call these technical units components. The whole system to be modelled is equally considered as a component and we say that the component "Controller System" contains the *sub-components* "Main Controller", "Auxiliary Controller" and "Power Unit". In Fig. 3-1 we show the component borders by dotted lines.

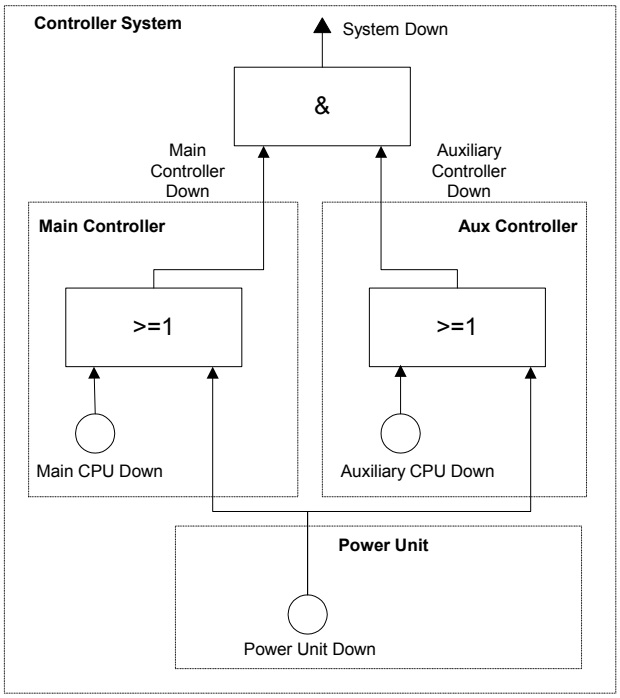

Fig. 3-1: Partition into Components

We see that the failure of either component Controller is an OR-term of one basic event that is generated within the same component and another basic event that is generated within a foreign component, namely the Power Unit. We assume that the model is *complete* for our purposes, i.e. there are no other failure influences than those represented by the Fault Tree edges. Then the only failure influences that cross component borders are:

- the two edges from the Power Unit to the Main Controller and to the Auxiliary Controller,
- the two edges from the Main Controller and the Auxiliary Controller to the upper OR gate belonging to the component "Controller System".

We make these interconnections between components visible by introducing *ports*, which are interfaces that allow joining subcomponents together. To preserve the direction of the gates we introduce two types of ports: input ports and output ports. Graphically we denote ports by solid triangles (see Fig. 3-2).

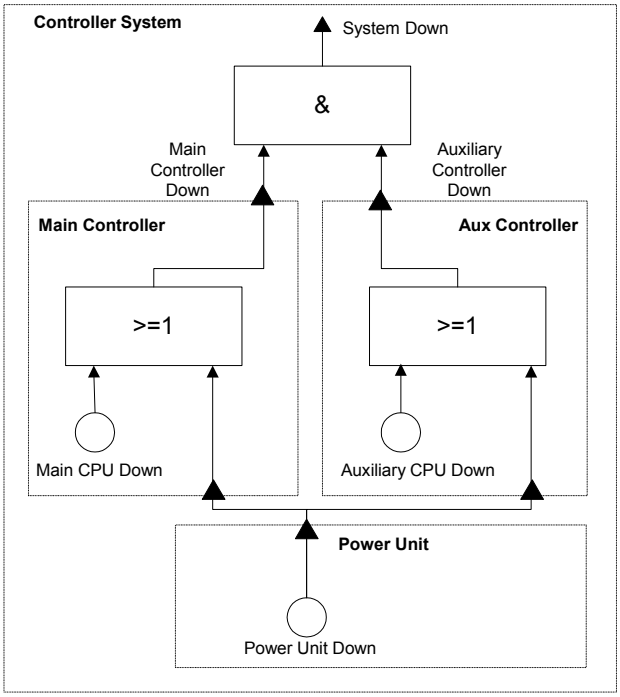

Fig. 3-2: Components with Ports

Ports can be the source or target of edges just as the ordinary Fault Tree nodes (basic events and gates). Note that the short edges joining the output port of component Power Unit to the input ports of component Main/Auxiliary Controller belong to the component Controller System while the edges drawn inside of each sub-component box belong to the respective component.

Since, by assumption, influences between different subcomponents only exist through the ports, we can now assert that every basic event inside of some component is stochastically independent of any basic event belonging to another component. This assertion can be relied on during the analysis of the complete graph.

As an additional benefit of the proposed component concept, it is now possible to store each component independently of each other and to have the components developed by different people. The developer of the Power Unit, for instance, makes an agreement with the developer of the Controller System about the ports of the Power Unit. Then both can continue their work independently: One person refines the internal structure of the Power Unit while the other is building the component Controller System around a black box representation of the Power Unit and both of the Controllers. The result of the system modeller's work looks like this (Fig. 3-3):

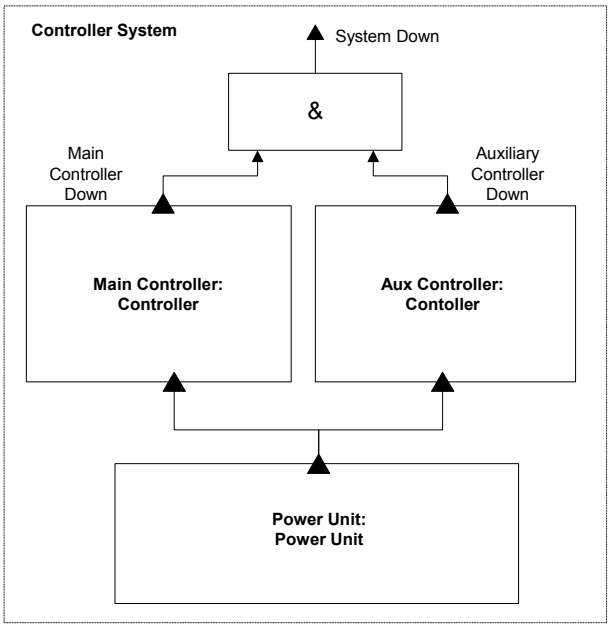

Fig. 3-3: The Component "Controller System"

The work of the Power Unit modeller looks like this (Fig. 3-4):

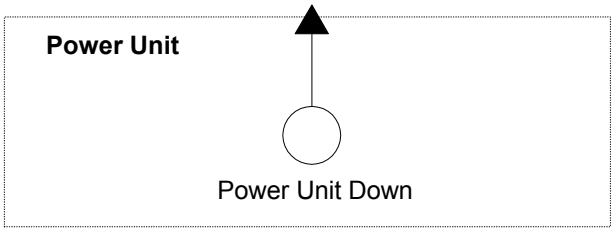

Fig. 3-4: The Component "Power Unit"

Each modeller can use all conventional Fault Tree elements, such as basic events and gates. As shown in this example, it is not mandatory that the top-level component contains only sub-components; standard gates and basic events may be used on system level as well.

Let us now assume that Main Controller and Auxiliary Controller are two devices of the same technical type. Thus, both of them show the same stochastic characteristics of their internal failure event "CPU Down". This does not mean that they always fail at the same time, but that the probability distribution and its parameters are the same for both. As a consequence, there is no need to model this component twice; a reuse is possible and even advisable. The model for the component type Controller is shown in Fig. 3-5. Two instances of this component are used within the Component Fault Tree model of the Controller System.

This example has revealed another advantage of our concept: Each component is modelled only once and reused as often as needed. The error-prone copy-and paste of parts of a Fault Tree has become obsolete. Note that if several instances of the component "Controller" are used in a system, each of it has an internal event named

"My CPU Down". However, this is not a repeated event, since all of these events are different and independent from each other. The common name just indicates that it is the same type of failure behaviour. If the modeller wanted to express a repeated event, it would have to be placed outside of the component (either inside of another component or on the next higher hierarchical level). Due to the Cause-Effect-Graph concept repeated events appear only once in the graph. In summary, private events of a component are not visible to any other component. In our tool, the combined identifier of component instance and event allows distinguishing repeated from individual events.

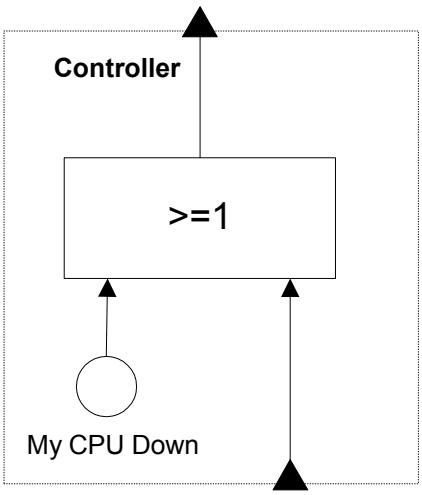

Fig. 3-5: The Component "Controller"

3.2 Analysing Component Fault Graphs

Mathematically, Component Fault Trees are described by a set of Boolean functions, each one belonging to one Output Port. Each function maps the input ports and the internal events of the component to a Boolean term assigned to an output port. To explain this facet of our concept, we start by considering a component with just one output port. To this end we revisit the Controller component from Fig. 3-5.

The Boolean formula represented by this component is

$$out_1 \Leftrightarrow in_1 \vee \text{MyCPUDown}$$

This formula gives a recipe how to calculate the probability of the output event from the probabilities of the internal event "MyCPUDown" and the probability of the Boolean term connected to the input port:

$$P(out_1) = 1 - (1-P(in_1))*(1-P(\text{MyCPUDown}))$$

Obviously, a quantitative analysis of the component Controller alone is not possible yet, since so far it is unknown what will be connected to the input port in_1.

If, in contrast, we examine the component Power Unit from Fig. 3-4, we will find that this time a quantitative Fault Tree Analysis is possible. This is due the fact that this component has no input ports. From this precondition for quantitative analysis follows that the system level component must not have any input ports. Components to be used as sub-components may have input and output ports. We see that if we embed one instance of the component "Power Unit" and two instances of the component "Controller" into the component Controller System from Fig. 3-3, the resulting system is a component without input ports and thus analysable.

Note that the information, what the ports are connected to, is not stored inside of the sub-components, but on the higher-level component "Controller System". Generally, the sub-components have no knowledge about their context.

Since we have extended Fault Trees to CEGs it is possible that components on any hierarchy level have more than one output port. For instance, the component from Fig. 2-4 has two output ports, modelling different failure modes that can be examined. If a component with several output ports is used within a higher-level system, this can introduce repeated events (in the example: CPU Down). These causal chains are resolved correctly due to the CEG concept. It is not mandatory that all existing output ports are connected or used for analysis. If the component on the highest hierarchy level has exactly one output port, then this output port corresponds to the top-event of a classical FTA; if it has more than one, each output port can be chosen as a starting point for analysis.

From this new point of view there is not much of a difference between components (that define arbitrary and often complex logical functions) and Fault Tree gates (that define elementary logic functions): for instance, a 2 out of 3 voter gate that many FTA tools offer can equivalently be described by a component "Voter 2oo3" with three input ports and one output port, as shown in Fig. 3-6:

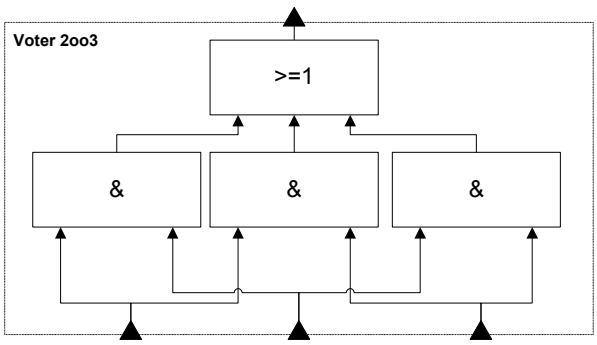

Fig. 3-6: A 2-out-of-3 Voter modelled as a Component

3.3 Formalisation

A Component Fault Tree (CFT) is a tuple (N, G, SC, E) consisting of

1. a set N of *Simple Nodes*, partitioned into
   a. a set N_{intern} of internal events
   b. a set N_{in} of input ports
   c. a set N_{out} of output ports
2. a set G of *Gates*, each of them described by
   a. one output port, denoted g.out

b. one or more input ports, denoted $g.in_i$, with $i \in \mathbb{N}$

c. a Boolean formula, (e.g. $g.out = g.in_1 \vee g.in_2$)

3. a set SC of *Sub-Components*, each one described by

   a. one or more output ports, denoted $sc.out_i$

   b. one or more input ports, denoted $sc.in_i$

   c. a mapping to another CFT

4. a set of directed *edges* $E \subseteq ((N_{intern} \cup N_{in} \cup G.OUT \cup SC.OUT) \times (N_{out} \cup G.IN \cup SC.IN))$ where

   - G.OUT is the set of all outputs of all gates,
   - G.IN is the set of all inputs of all gates,
   - SC.OUT is the set of all outputs of all sub-components,
   - SC.IN is the set of all inputs of all sub-components.

   The first element of the tuple representing an edge is called the *source* of the edge and the second element is called the *target*.

   As an additional restriction it is forbidden that two or more edges share the same target, i.e.

   $\nexists\ s_1, s_2, t: [(s_1,t) \in E \wedge (s_2,t) \in E \wedge s_1 \neq s_2]$.

♦

Some explanations:

- *Simple nodes* is the collective name for the internal events (basic events in traditional FTA) and the input and output ports belonging to the component being modelled. Internal events and input ports may be sources of edges; output ports may be targets of edges.
- Apart from simple nodes there are *gates* and *sub-components*. Both are very similar to each other, not only regarding the graphical representation as rectangular boxes. The differences are:
  - a gate has only one output port, whereas a sub-component may have more than one,
  - a gate has a Boolean function attached to it, whereas a sub-component has another CFT attached to it.

  Remembering that CFTs, by their graph structure, define Boolean functions from their inputs and internal events to their outputs, we find that both gates and sub-components represent Boolean functions (we know this from Section 3.2).
- Edges must not be connected directly to gates or sub-components but only to their input / output ports.
- Obviously the *own output ports* of a CFT can only be the target of edges while the output ports of the sub-components and gates being used can only be the source of edges and vice versa.
- The connection between different hierarchical levels is accomplished by two means:
  - by the mapping between a sub-component and its corresponding CFT
  - by the input and output ports that are joined by edges. The own input ports of the component currently being modelled appear as sub-component inputs on the next higher hierarchical level. The same applies to the output ports. The connection to the "right" port is assured by a combination of the unique component identifier and the unique port identifier.
- It is forbidden that a component contains itself as a sub-component - directly or indirectly. Violations of this rule (called "deep cycles") must be checked before analysis is started.
- Since the analysis relies on acyclic graphs it is further forbidden that there is any set of nodes and edges that forms a directed cycle (called "shallow cycle"). To avoid cycles across different hierarchy levels, edges leading from an output port of a sub-component to an input port of the same sub-component are forbidden, directly as well as indirectly. This must be checked before analysis is started.
- There is a restriction that two or more edges must not share a common target. The uniqueness of the edge source is essential when tracing the graph back during the analysis. This means that the edges could have been defined as functions (as opposed to relations) as well. Edges would then be defined in the style *source = f(target)*. However, for convenience we prefer the notation as a relation rather than as a function.
- It is however allowed for two or more edges to have a common source. This makes the difference between a DAG and a tree. The common source of two or more edges is semantically the same as a *repeated event*.
- Apart from the mentioned restrictions the user can group graph elements into components as appropriate.

3.4 Differences to Existing Approaches

The main difference between our approach and existing solutions is that we distinguish the subtree hierarchy determined by the logical structure from the decomposition hierarchy introduced by the system

architecture. The tree hierarchy leads from the root (the top-event) down to the leaves (the basic events), whereas the decomposition hierarchy leads from the whole technical system to the most detailed sub-components.

Fig. 3-7 demonstrates the kind of decomposition that many current FTA tools offer. It follows the first kind of hierarchy, from root to leaves. The analysis of a modular Fault Tree is done by merging the two partial trees at the "transfer ports" (IEC 61025) and then calculating the whole tree. This leads to the same result as first calculating the failure probability of the top event in "lower" and then replacing the connector in the component "upper" by a basic event with this probability. Note that component "upper" must carry the knowledge about which subtree is connected below and that a probability value can be assigned to the output of component "lower".

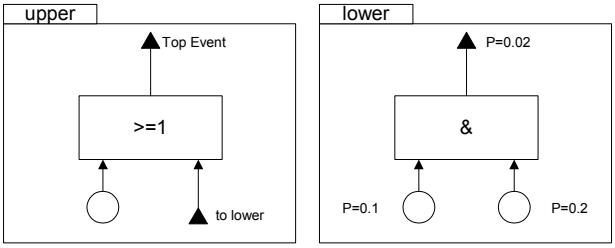

Fig. 3-7: Traditional Fault Tree Decomposition

By another example, we show the new kind of decomposition according to the second kind of hierarchy, see figure Fig. 3-8. This decomposition cannot be expressed in traditional tools, since the component "inner" is not a module. In spite of the common event the calculation leads to the correct result. This time, it is not the component "inner" that stores the knowledge about what comes below the input ports, but it is the component "outer" that keeps all edges leading to and coming from its sub-components. We do not assume that components are subtrees that can be simplified to one event with a fixed probability or rate, but rather treat them as Boolean functions.

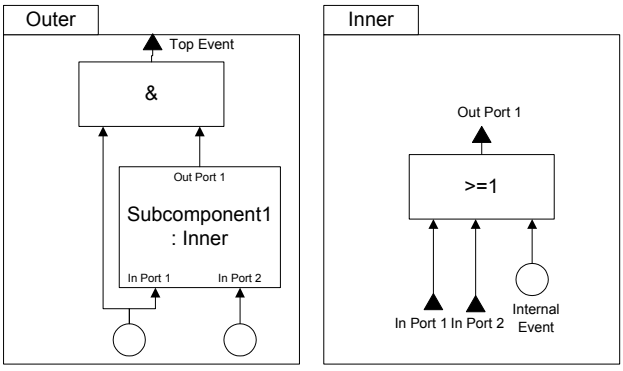

Fig. 3-8: Component-wise Fault Tree Decomposition

Focussing on this kind of hierarchy allows partitioning the system Fault Tree as appropriate to model real world components that influence each other and, vice versa, to build up a system Fault Tree out of existing Component Fault Trees. The expressive power supersedes what is available from commercial tools and the intuitive decomposition helps the modeller to build correct Fault Trees even for complex systems.

4 Practical Application

4.1 The Safety Analysis Tool UWG3

The Safety and Reliability Analyser tool UWG3 that has been developed in a co-operation between the Hasso-Plattner-Institute and the companies Siemens and DaimlerChrysler applies this new component concept. A screen-shot can be found at the end of this paper (Fig. 7-1). It shows the tool at work, modelling the Controller System Example shown above.

Its Windows based GUI offers different graph windows showing one Component Fault Tree each. At the right hand side there is a Component Explorer window that allows navigating through all open files, all components therein and all graph elements belonging to these components. Below there is a properties window that allows modifying functional parameters (e.g. probabilities) and style attributes (e.g. line colours and weights). Large amounts of numerical data are more comfortably edited in tables, so UWG3 offers a table view that allows data import and export to programs such as EXCEL. On the left side of the screen there is a repository of available graph elements, such as basic events or gates. The logical symbols displayed here are IEC 61025 style; they can alternatively be shown as international (US) symbols. Note that port symbols are offered in addition to the classical Fault Tree elements. Graph elements are applied by dragging and dropping them from the repository window into the graph window. The same mechanism is used to apply components as subcomponents of higher-level components: a component is dragged from the explorer window into another component window where it will appear as a black box, showing only the ports. Putting edges between graph elements generates the semantic connection.

The tool uses an open XML file format (Bray et al 2000) to store the component models. This facilitates later integration with other tools. Each file may contain one ore more components and the components belonging to one system may be distributed across different files. This permits models for different components to be edited concurrently.

Each graph element has a unique ID including the URI where the file is located. The tool automatically assigns and resolves the IDs, independently from the names that the user gives to events and components (the user can optionally display the internal IDs). This ID system also enables the detection of repeated events and serves as a cross-reference to other development documents, as proposed in IEC61025: graph elements are identical iff they have the same ID.

When analysis is started, all components belonging to the system are loaded. This is performed recursively in the sense that sub-components and their sub-components are searched and loaded until no more references are found. Consistency of the port signature is automatically checked. The loader guarantees that the model is

complete (no more open input ports) and free from cycles. The graph as a whole is transformed into one BDD and can then be analysed by the usual algorithms. The algorithms are taken from standard libraries. The fact that the graph actually consists of different independent parts is invisible to the analysis algorithm. Since a DAG can have, unlike a tree, more than one output port ("top-event"), it is necessary that the user specifies one for analysis.

4.2 Practical Application

Currently, we use the tool UWG3 to calculate reliability and safety figures for power plants and transportation systems. The implemented concepts for the usage of system components allow us to get more readable Fault Trees. So each person responsible for a system component is able to build the respective Fault Tree for a component. Afterwards the entire Fault Tree for the complete system is composed based on these component Fault Trees. This technique saves time and increases the quality of reliability and safety figures. In addition, it guarantees quick results due to implemented BDD-algorithm.

5 Conclusion and Future Work

In this paper we have proposed a new component concept for Fault Trees. The components need not be modules in the sense of classical FTA but may be partial graphs that correspond to real-world components. Input and output ports accomplish the connection of subcomponents to their context on the next higher level. Components that have no input ports can be analysed alone. Hereby, qualitative or quantitative analysis known from classical FTA is possible, e.g. by algorithms based on BDDs. Component Fault Trees that do have input ports cannot be analysed alone, but represent Boolean functions from input ports and internal events to the output ports. These components must be integrated into higher level components and appear there as sub-components (black boxes in the representation of our tool). Instead of trees, we have been using Directed Acyclic Graphs that we call Cause Effect Graphs for some time. This avoids the problem of repeated events by allowing more than one edge starting from any event or intermediate term and permits analysis of more than one failure mode.

Many researchers have stated that for today's complex embedded systems one single analysis technique is not sufficient, but the different techniques must smoothly integrate with each other (Bechta Dugan et al 1999), (Fenelon and McDermid 1993). Since we agree with this point of view, we are currently working on an integrated workbench, incorporating state/transition based probabilistic methods such as Markov-Chains and combinatorial models such as Fault Trees (Kaiser 2002). Our component concept makes an important step forward since it allows partitioning Fault Trees into independent components. The next step of our research efforts will be to integrate components that are described by Markov models. We plan to add a type system to input and output ports in order to ensure that only matching kinds of events or states can be joined together. The ID system and XML file format chosen for our tool is structured to cater for different kinds of future models.

Other issues in our further research are time-saving analysis techniques that simplify each component as far as possible and store the intermediate results along with the component data (e.g. reduced ordered BDDs). We are collaborating with other projects with the aim of automatically generating Fault Trees from technical data for Hardware and Software components and of annotating components with Component Fault Trees (Grunske 2003). In the future we plan to implement a component repository in order to store Component Fault Trees along with other technical data and to locate them when they are needed for analysis.

6 References

Bechta Dugan, J., Sullivan, K.J., Coppit, D.(1999). Developing a high-quality software tool for Fault Tree analysis. In *Proceedings of the Int. Symposium on Software Reliability Engineering*, p. 222-31, Boca Raton, Florida.

Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. (2000): Extensible Markup Language (XML) 1.0 (Second Edition) *W3C Recommendation* Accessed 14 July 2003

Bryant, R. (1986): Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677--691, Aug. 1986.

Clarke, E., Wing. J. (1996) Formal Methods: State of the Art and Future Directions, *CMU Computer Science Technical Report CMU-CS-96-178, August 1996*

Coudert, O. and Madre, J.C. (1993). Fault tree analysis: 10^20 prime implicants and beyond. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 240-5, Atlanta, GA.

DIN 25424 (1981/1990) *Fehlerbaumanalyse* (Fault Tree Analysis). German Industry Standard (Part 1 & 2). Beuth Verlag, Berlin.

Doyle, S.A. and Bechta Dugan, J (1995). Dependability assessment using binary decision diagrams (BDDs). In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 249-258, Pasadena, California.

Fenelon, P., McDermid, J.A., (1993), An Integrated Toolset For Software Safety Analysis. In *Journal of Systems and Software*, 21(3): p. 279-290

Grunske, L. (2003): Annotation of Component Specifications with Modular Analysis Models for Safety Properties. In *Proceedings of the 1st International Workshop on Component Engineering Methodology*, Erfurt, September 22, 2003

Harel, D. (1987): Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming 8(3)*: 231-274

IEC 61025 (1990). *Fault Tree Analysis* International Standard IEC 61025. IEC, Geneva

Kaiser, B. (2002): Integration von Sicherheits- und Zuverlässigkeitsmodellen in den Entwicklungsprozess Eingebetteter Systeme. In *Softwaretechnik-Trends* 22(4).

Kececiogliu, D. (1991), *Reliability Engineering Handbook* Part 1 and Part 2, Englewood Cliffs: Prentrice Hall

Kohda, T., Henley, EJ., Inoue, K. (1989): Finding Modules. in Fault Trees. *IEEE Trans. on Reliability*, Vol. 38, NO. 2, pp. 165-176

Liggesmeyer, P., Mäckel, O. (2001) Quantifying the Reliability of Embedded Systems by Automated Analysis. *The International Conference on Dependable Systems and Networks (DSN'01),* Goteborg, Sweden

Liggesmeyer, P.; Rothfelder, M. (1998): Improving System Reliability with Automatic Fault Tree Generation. In *Proceedings 28th Annual Fault Tolerant Computing Symposium,* Munich, June 1998, pp. 90-99.

Mäckel, O., Liggesmeyer, P (2000) Automatisierung erweiterter Fehlerbaumanalysen für komplexe technische Systeme. In *at-Automatisierungstechnik*, R. Oldenbourg Verlag, München, Februar 2000

Mäckel, O., Rothfelder, M. (2001): Challenges and Solutions for Fault Tree Analysis Arising from Automatic Fault Tree Generation: Some Milestones on the Way. *ISAS-SCI (1) 2001*: 583-588

Vesely, W. E., Goldberg, F. F., Roberts, N. H.,. Haasl, D. F.(1981) *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington

7 Appendix: Screenshot of UWG3

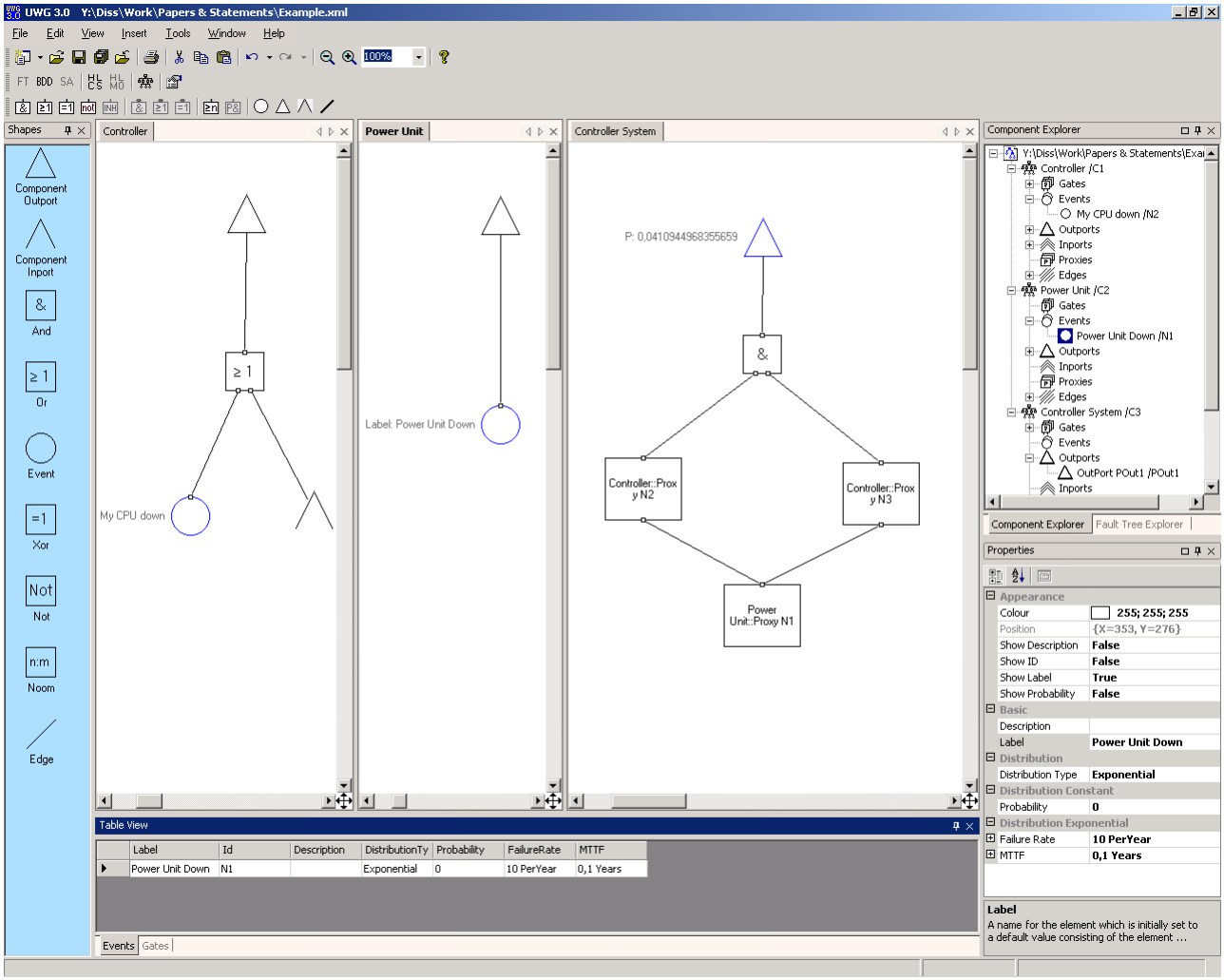

Fig. 7-1: Sreenshot from the tool UWG3